Async Support for TensorFlow Backend in FFmpeg

Organization - Intel Video and Audio for Linux
Applicant - Shubhanshu Saxena

Table of Contents

Basic Information

Name: Shubhanshu Saxena
Major: Computer Science and Engineering
Degree: Bachelor of Technology
Year: Sophomore
Institute: Indian Institute of Technology (BHU), Varanasi
Email Address: shubhanshu.saxena.cse19@iitbhu.ac.in
Alternate Email: shubhanshu.e01@gmail.com
GitHub: shubhanshu02
LinkedIn: shubhanshu-saxena
Resume: https://drive.google.com/file/d/1l27rC9gSlDYSJw5JhwCxilK55PV-aI8S/view
Telephone: +91-9166058795
Alternate Telephone: +91-8058002224
Timezone: Indian Standard Time (UTC+05:30)

Abstract

This project focuses on implementing an asynchronous mechanism for model inference and batch execution in the TensorFlow backend of the FFmpeg Deep Neural Network module to boost model inference performance.

The Tensorflow backend uses the TensorFlow C API, which currently does not provide functions for asynchronous execution. The support for async behavior can be provided using multithreading on the existing TensorFlow library functions. We will implement this behavior through detached threads that work independently of each other.

Several inference frames will be combined to a single input tensor and executed together in a single batch to enable the batch mode. The DNN module authors saw a performance gain in the OpenVino backend with asynchronous batch inference against synchronous inference. A similar performance gain is expected from this project.

Why Intel Video and Audio for Linux?

I am only applying to Intel Video and Audio for Linux for the GSoC 2021. Being a regular user of Linux, I have always wanted to contribute to some Linux projects. Using hardware acceleration to boost program performance has fascinated me since the day I started coding. The implementation of asynchronous behavior using threads made me think out of the box for something we usually don't see in our regular studies. I would love to contribute to something that makes me think unconventionally.

Deliverables

---

1. **Async Support in TensorFlow backend (Required)**
   Currently, the TensorFlow backend supports only the synchronous mode of model inference, which is single-threaded and slow. Using asynchronous mode in a multithreaded environment will provide us with a higher CPU utilization and faster execution due to its non-blocking nature.

2. **Async Support in the Native Backend (Optional)**
   The native backend is used for model inference when the target system does not support OpenVino or TensorFlow backend. This backend also currently supports only the synchronous model execution. We can also extend the async support in the TensorFlow backend using detached threads to the native backend.

3. **Support for Batch Mode in TensorFlow backend (Optional)**
   Loading multiple image frames as a single batch and inferring them at once is less expensive on the system than processing all frames one by one. Enabling batch inference for model inference will significantly boost the TensorFlow backend’s performance if clubbed with the async mode.

How is Performance Gained?

In the current scenario, the TensorFlow backend loads a single frame at once, processes it using the DNN model, and then returns the result. This step is repeated until all the frames have been processed. In other words, there is one inference request to the backend for each frame.

After completing this project, the backend will load many frames as a batch, process them using the DNN model in an asynchronous fashion without inter-dependency or blocking, and then return the resulting frames. A batch will contain only a single frame when the batch mode isn't supported. In other words, there will be one inference request to the backend for each batch. Each frame in a batch is considered as a task, so each request contains many tasks.

Due to this multi-threaded asynchronous approach, while processing a batch of AV Frames in a single inference request, we can significantly improve performance.

Asynchronous Execution in C

I intend to use the POSIX thread library to enable asynchronous execution by executing the TensorFlow sessions in separate threads. For this purpose, the POSIX library has two types of threads - Joinable or Detached threads.

1. Joinable Threads - If we use joinable threads, we need to join the newly created thread with the calling thread at the end of its execution. Hence, the calling thread's implementation is halted till the thread finishes its work, i.e., making the threads work synchronously.
2. Detached Threads - If we use detached threads, the newly created thread will work independent of the calling thread, and we wouldn't need to wait for its completion for the calling thread to resume its execution further, hence making the execution asynchronous.

Thus, we will use detached threads to implement the async execution in the following way. The comparison between these two execution mechanisms is available here.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
/**
* Parameters to be passed into the
* thread runner function
* */
typedef struct parameters
{
   pthread_t thread;             // threadId
   // Other parameters
   int (*callback)(void *param); // completion callback
} threadParameters;

/**
* Completion Callback for the thread function
* */
int completionCallback(void *param)
{
   // Some Callback Stuff
   return 0; // On Success
}

/**
* Runner Function for the thread
* */
void *runner(void *arg)
{
   // Parse the parameters for the execution
   threadParameters *param = (threadParameters *)arg;
   // Infer the model using TF_SessionRun
   // Call the completion callback
   param->callback(param);
}
/**
* Asynchronous Execution using Threads
* */
void asyncExecutor()
{
   // Create the Thread
   pthread_t thread;
   pthread_attr_t attributes;
   pthread_attr_init(&attributes);
   pthread_attr_setdetachstate(&attributes, PTHREAD_CREATE_DETACHED);
   threadParameters *tp = (threadParameters
*)malloc(sizeof(threadParameters));
```

```
    tp->thread = thread;
    tp->callback = &completionCallback;
    int returnValue = pthread_create(&thread, &attributes, runner, tp);

    if (returnValue != 0)
    {
        printf("Error Creating Thread!");
        exit(0);
    }
}
int main()
{
    asyncExecutor();
    asyncExecutor();
    pthread_exit(NULL);
```

Since the target system may not support the POSIX thread library, we need to handle those cases. FFmpeg handles these cases by checking if the ***HAVE_PTHREAD_CANCEL*** macro is present or not. So, we will add this check to the async execution function. In case POSIX threads aren't supported by the system, we will complete the execution to sync mode.

Support for Batch Mode in TensorFlow Backend

Currently, batch execution in FFmpeg requires the existence of an async execution mechanism for the backend. In the async mechanism without batch mode, one inference request to the backend triggers only a single task execution. But in batch mode, each inference request starts multiple task executions numbering equal to the batch size. When batch size equals 1, the way of execution is similar to the implementation without batch mode.

As of now, the FFmpeg DNN module uses the TensorFlow C API to infer on input tensor in NHWC format of the following dimension:

Dimension = [1, input height, input width, input channels]

The “1” here signifies the tensor contains the images in the batch of 1. In other words, a single image per tensor. In batch mode, this dimension changes to the ***batch_size*** i.e., a single tensor contains multiple images.

I plan to add the batch mode to the TensorFlow backend by reusing the same request-based mechanism with the async method. Unless provided, we will set the batch size to 1. Each request

will be executed only when it contains tasks amounting to the batch size in number. On inference, we will fill the tensor data with data of all images in the batch and then infer the request asynchronously. In this way, multiple tasks are being executed simultaneously in the same request, while numerous requests are being executed asynchronously in detached threads.

For example, if we want to apply a DNN filter on a video with 66 frames in batches of 32. We will execute the first 64 images in batches of 32, but the last two images will be left out. In this case, we infer the remaining two images asynchronously one frame at a time. This is called flushing, and we will use the function ***ff_dnn_flush_tf*** for this purpose. The Implementation Plan section contains a section below for necessary changes in the async mode to support batch processing.

Implementation Plan

Like the OpenVino backend, the TensorFlow backend needs to be switched to using the Request and Task queues to infer the Deep Neural Network models.

This approach mainly helps handle the asynchronous cases where several requests (numbering to the '***nireq***' option) are stored in a Request Queue used to execute requests. Following this approach in the TensorFlow backend will also easily unify the three backends in the future when the async mode is supported in all three backends.

I am planning to complete this project's implementation by the following changes to the TensorFlow backend:

1. Switching Execution to Request-Task based Mechanism by refactoring the existing code
2. Adding async function ***ff_dnn_execute_model_async_tf*** for the backend.
3. Adding the function ***ff_dnn_get_async_result_tf*** to the backend for getting async results.
4. Adding a completion callback function to get the resulting frame.
5. Adding functions to infer a ***RequestItem*** asynchronously.
    a. tf_infer_request_infer for synchronous inference
    b. tf_infer_request_infer_async for asynchronous inference
    c. tf_infer_request_free for freeing the infer_request
6. Modifying the async functions to support batch mode.
7. Creating a function ***ff_dnn_flush_tf*** for flushing the extra frames in batch mode.

Defining the Data Types for Request-Task based Mechanism

The structures TFOptions and TFModel need to be changed in order to add async execution queues and a 'number of requests' options to the TensorFlow backend. At the end of this step, these structures will have the following definition:

```
typedef struct TFOptions{
   char *sess_config;
   int nireq;
} TFOptions;

typedef struct TFModel{
   TFContext ctx;
   DNNModel *model;
   TF_Graph *graph;
   TF_Session *session;
   TF_Status *status;

   SafeQueue *request_queue;
   Queue *task_queue;
} TFModel;
```

The ***request_queue*** is a SafeQueue instance, a queue with mutex locks to keep the queue's consistency when it is accessed by multiple threads simultaneously, while the ***task_queue*** is a simple Queue used to store the task items. Since the execution order may not remain the same in async mode, the ***task_queue*** is necessary to retain the order of image frames in the video. The RequestItem has a list of TaskItems containing the frames to be inferred in that batch. In all, a RequestItem is the basic unit for inference, while a TaskItem includes details about a single image frame.

To switch to a request-based inference mechanism, we define the data types ***RequestItem*** and ***TaskItem***. The total number of RequestItems in an async inference call remains constant and equal to ***nireq*** defined in the TensorFlow backend options. The RequestItem contains the task it is currently executing while the TaskItem contains all the inference task details. These are defined in the following way:

```
typedef struct TaskItem {
   TFModel *tf_model;
   const char *input_name;
   AVFrame *in_frame;
   const char *output_name;
   AVFrame *out_frame;
   TF_Tensor output_tensors;
   int do_ioproc;
   int done;
   int async;
} TaskItem;
```

```
typedef struct RequestItem {
   tf_infer_request *infer_request;
   TaskItem **tasks;
   int task_count;
   int (*completion_callback)(void *args);
} RequestItem;
```

Loading the TensorFlow model

The first call to the TensorFlow backend is on the ***ff_dnn_load_model_tf*** function, which loads the TensorFlow graphs from the model file and initializes the DNN modules necessary for inference.
The function has the following declaration:

```
DNNModel *ff_dnn_load_model_tf(const char *model_filename, DNNFunctionType
func_type, const char *options, AVFilterContext *filter_ctx)
```

Pseudocode for the updated function:

1. Allocate space for ***DNNModule*** and ***TFModel***
2. Set default options using ***av_opt_set_defaults.*** Or parse options from the model context using ***av_opt_set_from_string.***
3. Load TensorFlow graphs using ***load_tf_model***. Or try loading the native model using ***load_native_model***.
4. Set ***nireq*** i.e., number of async inference requests to the default value if not provided. The default value is a rough estimation in the OpenVino backend equal to ***cpu_count/ 2 + 1***.
5. Initialize the request queue and task queue
6. Create inference requests for asynchronous execution and push them to the back of the ***request_queue*** of the model.
7. Return the created ***DNNModel*** instance.

Modifying the synchronous execution function

For synchronously inferring the TensorFlow model, the function ***ff_dnn_execute_model_tf*** is used with the following declaration.

```
DNNReturnType ff_dnn_execute_model_tf(const DNNModel *model, const char
*input_name, AVFrame *in_frame,const char **output_names, uint32_t nb_output,
AVFrame *out_frame)
```

For using the same key code for both async and sync mode, this function calls another function, ***execute_model_tf,*** which runs the inference on the ***TF_SessionRun.***

Pseudocode for ***ff_dnn_execute_model_tf:***

1. Create an inference ***RequestItem*** and ***TaskItem***.
2. Check if the input frame and the output frames are null. If yes, return ***DNNError***.
3. Set the task->async to 0.
4. Set all the function parameters in the ***TaskItem*** and assign the task to the ***RequestItem***.
5. Execute the task using ***execute_model_tf*** with the created request.

Adding the asynchronous execution function

This function creates a new ***TaskItem*** and initializes it with the task parameters. Now, it pushes it to the end of the ***task_queue*** of the model. Next, it pops the front of the ***request queue*** from the model and adds this task to the ***tasks*** array in the extracted request. Finally, it calls the ***execute_model_tf*** on this request.

This function will have the following declaration:

```
DNNReturnType ff_dnn_execute_model_async_tf(const DNNModel *model, const char
*input_name, AVFrame *in_frame, const char **output_names, uint32_t nb_output,
AVFrame *out_frame);
```

Pseudocode for **ff_dnn_execute_model_async_tf**

1. Check if the input frame and the output frames are null. If yes, return ***DNNError***.
2. Create a new ***TaskItem*** and initialize it with the task parameters.
3. Set the ***async*** to 1 in this ***TaskItem.***
4. Push this task to the back of the ***task_queue*** in ***TFModel***.
5. Pop the front of the ***request_queue*** in the ***TFModel***.
6. Assign this task to ***request->tasks[request->task_count++]***.
7. Execute the popped request with ***execute_model_tf*** asynchronously.

Modifying Common Execution Function

All three backends use a common execution function to reuse the same key code for both async and sync execution. In the case of the TensorFlow backend, the function ***execute_model_tf*** is used for common execution with the following declaration:

```
DNNReturnType ff_dnn_execute_model_async_tf(const DNNModel *model, const char
*input_name, AVFrame *in_frame, const char **output_names, uint32_t nb_output,
AVFrame *out_frame);
```

Pseudocode for ***execute_model_tf***

1. Async Mode
    a. Fill model input tensors and input operations
    b. Set inference request and completion callback

    c. Start async inference request

2. Sync Mode
    a. Fill model input tensors and input operations
    b. Set infer request and start inference
    c. Call the inference completion callback

Adding function to fill Model Inputs

This function ***fill_model_input_tf*** takes the TensorFlow model and the request instance to fill the ***input_tensors*** and ***tf_input*** in ***tf_infer_request*** required by ***TF_SessionRun*** for model inference. It will have the given definition:

```
static DNNReturnType fill_model_input_tf(TFModel *tf_model, RequestItem
*request)
```

Pseudocode for ***fill_model_input_tf***

1. Use the existing function ***get_input_tf*** to get the TensorFlow operation to be performed and set the height, width, and input channels from the Tensor shape.
2. Set ***DNNData*** input object's height and width from input frame's height and width
3. Set ***graph operation*** in ***tf_input*** by its name using ***TF_GraphOperationByName***
4. Allocate input_tensor using the existing function ***allocate_input_tensor***.
5. If ***do_ioproc*** is 1, preprocess the input frame data to the ***DNNData*** input using the pre_procp function in the model or use the default function ***ff_proc_from_frame_to_dnn***.
6. Allocate space for ***tf_outputs,*** i.e., an array of ***TF_Output*** items and ***output_tensors,*** an array of TF_Tensors.
7. Iterate over each of the ***TF_Output*** items and assign it its graph operation, a TF_Operation instance, by its name.

Adding Inference Callback function

This function ***infer_completion_callback*** creates a local ***DNNData*** variable for output and sets the output from the ***output_tensors*** TensorFlow inference request. Next, it transfers the DNN data to the output frame. Finally, it deletes all the allocated tensors and cleans up all resources.

```
static void infer_completion_callback(void *args)
```

Pseudocode for ***infer_completion_callback***

1. Parse the ***RequestItem*** from the ***args***
2. Create a ***DNNData*** instance for output and set its parameters from the ***output_tensor*** in the ***RequestItem***.

3. If do_ioproc is 1, post-process the output frame from the ***DNNData*** using ***post_proc*** functions if present in the model or ***ff_proc_from_dnn_to_frame***. Else, set the output frame height and width from the ***DNNData*** object.
4. Clean up the allocated resources that won't be used after inference completion, like ***output_tensor***, ***input_tensors***, and ***tf_output***.

Completion Callback for the Asynchronous Execution

For the synchronous model inference, we already used ***infer_completion_callback*** as a callback on successful model inference. Since the asynchronous inference runs on a detached thread, we need to assign it a completion callback in its RequestItem instance.

Adding a Function to flush Extra Frames (Batch Mode)

The function ***ff_dnn_flush_tf*** will be used to flush extra frames that do not fit the inference batch size. In this case, we will infer these frames asynchronously using the functions we defined for the async mode. It will have the following definition:

```
DNNReturnType ff_dnn_flush_tf(const DNNModel *model)
```

Pseudocode for ***ff_dnn_flush_tf***

1. Pop a ***RequestItem*** from the ***request_queue*** of the ***TFModel***.
2. If the task count is 0, push the request back to the queue and return since no pending task is left to flush.
3. Fill the ***input_tensors*** and ***tf_input*** in ***tf_infer_request*** required by ***TF_SessionRun*** for model inference using ***fill_model_input_tf***.
4. Set the completion callback
5. Infer the request asynchronously.

Asynchronous Execution of Requests

Each ***RequestItem*** contains a pointer to a ***tf_infer_request***. Each ***tf_infer_request*** contains the parameters for the ***TF_SessionRun*** required for the execution. This data type is defined as follows:

```
typedef struct tf_infer_request {
   TF_Output* tf_input;
   TF_Tensor* input_tensors;
   TF_Tensor* output_tensors;
   TF_Output* tf_output;
} tf_infer_request;
```

For async inference of ***RequestItems***, we need to define the following functions:

1. Synchronous Inference - The function ***tf_infer_request_infer*** will be used for the sync mode execution. It takes the ***RequestItem*** as a parameter and calls the ***TF_SessionRun*** in the same calling thread with parameters from the RequestItem and its peek task.
2. Asynchronous Inference - The function ***tf_infer_request_infer_async*** will be used for the async mode of execution and takes the ***RequestItem*** as a parameter. It calls the ***TF_SessionRun*** in a separate detached thread and calls the ***request->completion_callback*** function on successful completion of ***TF_SessionRun***.

3. Freeing the Infer Request - After completing ***tf_infer_request_infer_async***, the output frame has been written from the ***output_tensors*** and ***tf_output***. These are no longer needed for model inference and need to be freed. The function ***tf_infer_request_free*** frees all the allocated resources in the ***request->infer_request*** using the functions ***TF_DeleteTensor*** and ***av_freep***.

Note: The same function ***infer_completion_callback*** is used as the completion callback of the ***infer_request*** to ensure the same key code isn't repeated.

Async Support in the Native Backend

The native backend has a similar approach to model inference as that of the TensorFlow model. First, loads input operations and data into ***DnnOperand*** and ***DNNData*** instances. Like the TensorFlow backend calls the ***TF_SessionRun***, this backend calls the ***pf_exec*** functions on individual model layers in a loop to infer the model. Finally, it fetches the output frame from the model output data.

These layer operations can also be run in detached threads for each frame to support the asynchronous mode. To keep the code's readability, we will change this model inference to a request-based mechanism as we will do for the TensorFlow backend through refactoring of the existing code.

Batch Mode

After implementation of async mode in the TensorFlow backend, we will require the following changes to the existing code:

1. Add ***batch_size*** to the ***dnn_tensorflow_options*** and ***TFOptions*** in the TensorFlow backend.
2. By **default**, set ***batch_size = 1*** unless not provided.
3. In ***allocate_input_tensor*** function:
    a. Change input tensor dimensions to support batch inference

```
input_dims[] = {batch_size, input->height, input->width, input->channels};
```

    b. Change the allocation size of input tensor

```
TF_AllocateTensor(dt, input_dims, 4, input_dims[0] * input_dims[1] *
```

```
input_dims[2] * input_dims[3] * size);
```

4. Remove ***av_assert0(dims[0] == 1)*** in ***get_input_tf*** function.

To support batch inference, we will modify the async execution function ***ff_dnn_execute_model_async_tf*** in the following manner:

1. Create and add the task to request_queue.
2. Then,

```
if (request->task_count == batch_size) {
    execute_model_tf(request);
}
else {
    ff_safe_queue_push_front(tf_model->request_queue, request);
    return DNN_SUCCESS;
}
```

In ***execute_model_tf*** function, for async mode:

```
if (task_count < batch_size) {
    return DNN_ERROR;
}
else {
    float* tensorData = (float *)TF_TensorData(input_tensor);
    for (int i = 0; i < batch_size; i++) {
        // fill tensorData[i] using DNNData of ith frame
        // This is possible because TF_Tensor holds a multidimensional
        // array of elements of a single data type.
    }
}
```

For the ***infer_completion_callback***, iterate over all tasks (***let index = i***) and set the output to the tensorData[i]. In other words, we are writing all frames from the output tensor.

Project Timeline

Community Bonding Period May 12, 2021 - June 7, 2021

- I will review the existing filters and experiment with their TensorFlow implementation to better understand how they work.

- I will discuss the existing code for async and batch mode in the TensorFlow and Native backend with the mentors.
- I will discuss the latest changes in the DNN module with the mentor and plan the execution accordingly.
- I will review the TensorFlow C API and carefully study it for error handling to avoid memory leaks in the execution.

Week 1 June 7, 2021 - June 13, 2021

- I will start implementing the TaskItems and RequestItems and refactoring the existing code to switch the execution.
- I will start refactoring the existing functions in the TensorFlow backend to use the newly defined data types.

Week 2 June 14, 2021 - June 20, 2021

- I will complete the loading, synchronous execution, and freeing the model part of the refactoring.
- I will start defining functions for infer_request and inference callbacks in the TensorFlow backend.

Week 3 June 21, 2021 - June 27, 2021

- I will complete the infer_request functions in the backend and integrate them for synchronous execution.
- I start adding functions for async execution and async result fetching.

Week 4 June 28, 2021 - July 4, 2021

- I will add the functionality to save and read the async result from the inference
- I will complete the async execution for the TensorFlow backend.

Week 5 July 5, 2021 - July 11, 2021

- This week is devoted to finding memory leaks and debugging the code for errors.
- I will reexamine the inference mechanism for any flaws

Phase 1 Evaluations July 12, 2021 - July 16, 2021

- We can deploy the Asynchronous Inference Mode.

Week 6 July 12, 2021 - July 18, 2021

- I will start implementing the Request-based inference mechanism and refactoring the existing code in the Native backend similar to the TensorFlow backend.

Week 7 July 19, 2021 - July 25, 2021

- I will add separate functions for async execution and the detached thread functions to execute RequestItems in the Native Backend with async mode.

Week 8 July 26, 2021 - Aug 1, 2021

- I will complete the async execution for the Native backend.

Week 9 August 2, 2021 - August 8, 2021

- I will start refining the existing code to support batch mode and change tensor dimensions based on the batch size.
- I will start modifying the ***ff_dnn_execute_model_async_tf*** function to call the execution function conditionally.

Week 10 August 9, 2021 - August 15, 2021

- I will start modifying the ***execute_model_tf*** function to fill the tensor with all images' data in the batch.
- I will start working on the flushing function for extra frames and complete the remaining work to add the batch mode.

Final Evaluations August 16, 2021 - August 23, 2021

- Batch Mode can be successfully deployed

About Me

Personal Details

My name is Shubhanshu Saxena. I am a sophomore in the Department of Computer Science and Engineering at the Indian Institute of Technology (BHU), Varanasi. Since my first year of college, I have been into coding and have grown my interest in exploring different fields of computer science.

I am also an active contributor to open-source since last year. My open-source contributions can be viewed here. I have been trying out various technologies ranging from web development, mobile development to machine learning. Over the last few months, my interest has shifted towards deep learning.

Apart from the academic stuff, I am an active member of the Developer's Group in the Club of Programmers, and I am a table tennis player. I have represented my institute in Udghosh 2019 in the Table Tennis team. Recently, I participated in the Traffic Sign Recognition competition for the

UI team in Inter IIT Tech Meet and worked in the tech team of Technex. I keep on exploring my interests in different fields through participation in various competitions.

In terms of my technical skills, I'm proficient in C++, Python, JavaScript, Dart and have basic knowledge in C, but I haven't applied it in any large-scale project yet. I am a keen learner and ready to learn anytime. That's why I feel I can compensate for this. I am also proficient with Git. I have been practicing TensorFlow for a few months now in my Artificial Intelligence course labs and have had essential multithreading experience from the Operating Systems course.

Communication

Over the summers, I will be available to work usually around 2 pm to 1 am IST on weekdays though I can be flexible with my schedule as per the mentors' availability. I may not be available for a few days due to some family trip, but I'll inform the mentors about that beforehand. I would love to spend time with the mentors and the team learning from them over the summers.

I am comfortable with any means of communication - be it email, voice call, or video meetings, as per the mentors' convenience. For the language part, I speak English and Hindi.

Post-GSoC

Since this project is vast in terms of scope, some of the points mentioned may be left unimplemented when we include the optional parts. If such a case happens, I will try to complete them after GSoC. After the GSoC, I would like to keep contributing to the DNN module and explore other Intel Open Source projects.

Development Environment

I use Linux as my primary operating system, though I have access to Windows 10 as well. Currently, I have Ubuntu Budgie 20.10 installed on my pc as the primary operating system. I have successfully installed FFmpeg from the source code on my laptop.