Multi Threaded BLOB Analysis for Video Processing

Matthew Hait
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: mhait@oakland.edu

***Abstract*—**This paper describes the design and implementation into incorporating a multi-core application for binary large object (BLOB) analysis in an embedded environment. The methodology uses parallelization on discrete function level, as well as pipelining the entire image calculation process. This paper presents a method into extracting image BLOB data it's optimal parallelization strategies.

I. INTRODUCTION

Unlike thermocouples, which measure a single temperature with a slow transient response, thermal imaging allows for measuring thousands of temperature readings across an image plane simultaneously. To extract measurable data from thermal video, BLOB tracking may be implemented to isolate temperature readings of an object from background noise. Further image processing can be used to extract average temperature and centroid location.

The long processing time associated with these image convolutions results in thermal imaging not being directly applicable for stimulus in an embedded real-time system. This paper explores the implementation of various multithreading methods through the oneAPI Threading Building Blocks (TBB) to reduce the computation time of BLOB tracking and image convolutions within the frame time of the source video. The software will extract BLOB centroid and average temperature values.

II. METHODOLOGY

In the software's execution, there are three main processes: loading data, frame calculation, and output data saving.

A. Loading Data

The software allows for three input file types: *.jpg, *.bin (binary file), and *.mov video files. Processing these file formats are supported by the FFmpeg libraries [1] and stb libraries [2]. These libraries are installed using a package manager and linked to the project through CMake. The image and binary file inputs permit single frame and batch image processing.

During batch processing and parallel pipelining, loading the images was included in the processing times recorded. The same image loading function was used in sequential and parallel pipeline batch loading in a sequential format to make the timing data comparable.

B. Frame Processing

The first step of processing the frame is to translate the frame to grayscale if needed. This is done using ITU-R BT.601 recommendation luma constants [3]. For the most accurate measurements, the image frames should be recorded into single channel grayscale.

The image is then copied into a Boolean image under a masing operation. Any pixels from the translated grayscale image whose pixels are above a temperature threshold set by the user correlate to setting the Boolean pixels true. This masks the background of the images from being used in future functions. To parallelize masking, TBB parallel_for was implemented subdivide the image into chunks that are given to other threads to process.

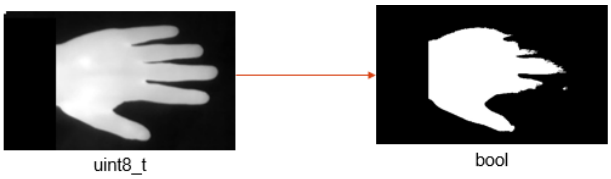

Fig 1. **Image masking**

The Boolean image is then processed by the recursive grass fire algorithm for image BLOB recognition [4]. The following connectivity kernel is in the grass fire algorithm:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig 2. **Recursive Grass Fire contact kernel**

The center of the kernel is placed at the first pixel of the boolean image that is true. The algorithm then loops through the contact kernel. When a boolean pixel value of "true" overlaps with a kernel value "true", the pixel on the boolean mask is burned (set to false). That pixel is then assigned the current BLOB id in a mapping image. The algorithm is then re-ran recursively with the kernel recentered on the pixel just

burnt. After recursion ceases, the BLOB id is incremented and the algorithm is repeated until all pixels in the mask image are set to false.

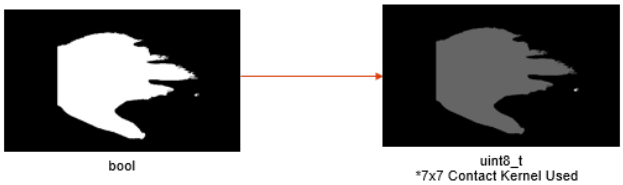

Fig 3. **Recursive Grass Fire transformation**

An attempt to parallelize the algorithm was done using TBB task groups. This implementation suffered from issues in software concurrency and ultimately was slower than the sequential implementation.

Next, the software processes the map and grayscale image into a BLOB average function. This function iterates through BLOB IDs, finds the pixels in the image map with the corresponding ID, and grabs the value of the coresponding pixel in the grayscale image. The values are added and the summation is divided by the number of pixels in the BLOB to find the average grayscale pixel value of the BLOB.

To parallelize this function, two methods were tested. The first method used TBB parallel_for to split BLOB IDs into separate threads. This implementation has the best performance when an image contains many BLOBS. The second approach was to use TBB parallel_for to split BLOB IDs into separate threads, and then use TBB parallel_reduce to calculate the average per BLOB.

The software then takes the BLOB map image and processes it through the centroid function. This function finds the centroids of the BLOBs by averaging the X and Y values of every pixel assigned to each BLOB. The parallelization stratigy used was similar to the BLOB average value function. The first method used TBB parallel_for to split BLOB IDs into separate threads. Each thread is then processed the average X and Y value for it's BLOB ID. The second approach was to use TBB parallel_for to split BLOB IDs into separate threads, and then use TBB parallel_reduce to calulate the average X and Y per BLOB.

The frame processing algorithm is further multithreaded by running each frame in a TBB Parallel Pipeline. This creates individual threads for each image and allows parallelization for some of the functions which couldn't be paralyzed internally.

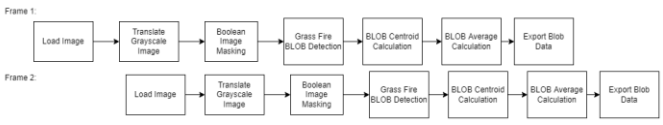

Fig 4. **Program parallel pipeline flow**

III. Experimental Setup

To measure the multithreading time savings, a benchmark function was added to the program. This function generated 100 white images with a resolution ranging from 76x43 to 7680x4320 pixels. The function would then record the compute time for every step in the pipeline sequential and parallel. By making the images entirely white, any variations in the output timing are from pixel count and not BLOB count.

The software was also implemented with a compare option. When enabling the compare function, the software would compute using sequential and parallelization to compare the results. This could be used for examining the time savings during single image and batch processing.

IV. Results

In parallelizing the masking function, up to 50%-time savings was witnessed on a four core I5-7600 for images ranging in size from 1232x688 to 7392x4128 pixels. However, when the function was run on an Intel Atom N2600, the parallel function had the exact same processing time as the sequential. This could be the result of the process scheduler on the Atom scheduling the parallel tasks on the same core as the parent task.

The parallel recursive grass fire function was slower than the sequential and worsened as the number of pixels in BLOBs increased. This function was then excluded from the parallel pipeline. The BLOB average and centroid functions were both found to be fastest using the TBB parallel_for only method. The parallel_for and reduce method can be faster if a high number of BLOBs are found in the images.

Finally, using a series of 500 576x343 images, the parallel pipeline was found to be 50% faster than the sequential batch method. Most of the time savings are the result of the individual parallelization of each function. Increasing the batch quantity from five images to 100 images did not show a dramatic increase in performance for the parallel pipeline as seen in Fig 5.

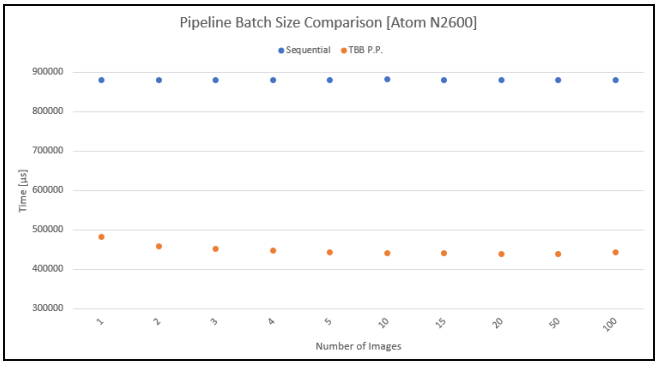

Fig 5. **Pipeline batch size comparison**

The parallel pipeline reduced the computation time a further 50% when moving from two cores to four cores on the Intel Atom N2600. To find the limits of optimization from parallelization, the program was moved onto an AMD 1700X 16 threaded processor. In testing 500 576x343 images, the limit of optimization was found to settle at 20ms per frame.

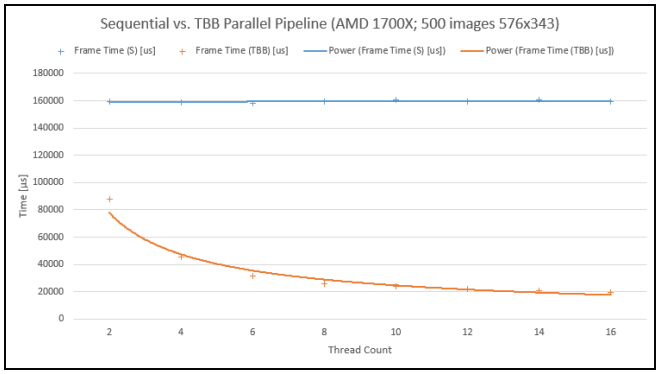

Fig 6. **Sequential vs. Parallel Pipeline**

CONCLUSIONS

The recursive grass fire algorithm is not sufficient in the manner that it was implemented. Faster processing could be done by first performing closing and opening filters and then using a smaller contact kernel. Ultimately, there are more parallelization strategies that could be tested, and optimizations to be discovered. To reduce the processing time below the frame time of a video source, parallelization is clearly a requirement. Threaded Building Blocks appear to be the best method to incorporate parallelization as it usually simplifies issues with software concurrency and simplifies the multi-threading strategy.

REFERENCES

[1] FFmpeg et al, “FFmpeg” Github. 19-Sep-2021. Available: https://github.com/FFmpeg/FFmpeg

[2] Nothings et al, “STB,” Github. 10-Sep-2021. Available: https://github.com/nothings/stb

[3] ITU-R, rep. Available: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf

[4] T. B. Moeslund, “CH7. BLOB Analysis,” in Introduction to video and image processing: Building real systems and applications, London: Springer, 2012, pp. 103–107.