CSE 237D Final Report - Baboons on the Move

DEBADITYA BASU, SANANYA MAJUMDER, ZHI WANG, BAIQIANG ZHAO

Baboons are intelligent, social animals that live in troops, and scientists are interested in studying their group-level decision making. The Baboons on the Move project helps scientists study the social dynamics of a troop of baboons in Kenya, using drones to record baboon movement and activities. Video footage from the drones are then post-processed to track individual baboons using computer vision technology. However, the current implementation takes a huge amount of time for post-processing, and one of the main reasons is that the implementation uses Python, which is an interpretive language. It remains unclear whether implementing the algorithms in a machine-level language, such as C++, can significantly improve the runtime performance. In this project, to speed up the runtime of post-processing, we complete an implementation in C++ and investigate the feasibility of parallelizing certain algorithms and implementing them in CUDA. We have achieved a speedup of *30x* as compared to the current Python implementation and set up a testing framework to validate the output results of the C++/CUDA implementation, so that post-processing can be completed in a reasonable time frame without compromising on the quality of results.

Key Words: Motion Tracking, Image Restoration, CUDA Programming, Benchmarking, Speedup

1 INTRODUCTION

The behavior of monkeys and apes has always held great fascination for scientists considering that they have a lot in common with humans, and baboons especially live in groups and display complex behavior [14, 16]. The Baboons on the Move project [1, 3] has been ongoing for many years and provides a system to understand the social relations and dynamics of baboon troops located in Kenya.

Specifically, the movement and activities of a troop of baboons are recorded using drones and post-processed using computer vision algorithms. The algorithms track their movements on all kinds of terrains along with predicting their movements if they are hidden by obstacles. As a result, the algorithms involve multiple stages and take a huge amount of time for post-processing. For example, for a video footage that is only 1-minute long, post-processing can take hours to complete on an average laptop computer. This makes run-time a significant bottleneck for the project.

As the current codebase is implemented in Python, one potential way to speed up post-processing is by implementing the core functionalities in a faster language like C++ and CUDA. C++ is a machine language and statically typed, whereas Python is an interpreted language and dynamically typed, and hence the code is interpreted at runtime instead of being compiled to machine code at compile time. CUDA, on the other hand, provides a platform for parallel execution of the code on GPUs and hence is much faster than both C++ and Python. The parallel execution is supported by the single instruction multiple threads (SIMT) architecture of the GPUs, which supports execution of the same instruction on different parts of the data by threads in parallel. This essentially speeds up any computation where the same operation can be applied to different parts of input independently.

Our goal of the project for this quarter was to complete an implementation of the post-processing algorithms in C++ and CUDA to improve on the performance while ensuring that functionality is not compromised. To achieve this, we

- Inherited some existing code and finished porting the algorithms to C++ and then CUDA.
- Created a testing framework and ensured functionality of the C++ and CUDA implementations. We were able to match the output to within *70%* with the C++ implementation and *70%* with the CUDA implementation.
- Created a benchmarking framework and showed that we were able to achieve a speedup of *2x* with the C++ implementation and *30x* with the CUDA implementation.

2 RELATED WORKS

Baboons. Scientists have been studying the behaviors of baboons for decades, and in particular, their group-level decision making and social dynamics within a troop [6, 17]. These studies are typically conducted based on first-hand observations by in-field researchers (e.g. [13]), and one disadvantage is that these observations often only provide a partial view of a troop of baboons [1, 3]. The Baboons on the Move project aims to provide a different perspective, namely, a global view, by using a drone to take footage of an entire site and post-processing the video footage using motion detection algorithms [1, 7].

Motion detection of animals. As discussed in [1], motion detection of individual animals from videos taken by an unstable camera far away from the subjects has been studied by [5]. However, the Baboons on the Move project focuses on tracking moving subjects as a group and studying their group-level behaviors. In general, motion detection of animals has been studied in previous works. For example, Sridhar et al have developed a tool named Tracktor that can track single animals under noisy conditions [12]. Zootracer [10] is a software developed by Microsoft Research, which allows users to provide any video recording as an input and tracks animal movements. Walter and Couzin have proposed a multi-animal tracking system, TRex, and evaluated the runtime and robustness of the system on animals such as termites and fruit flies [15].

Runtime comparison of programming languages. In this project, we complete a C++/CUDA implementation of the same algorithms and compare the runtime performance of it with the existing Python implementation. We note that general comparative analysis of different languages has been done by many researchers [8, 18], and runtime of Python and C++ has been compared, for example, on the N-Queens problem [4]. For CUDA and GPU programming, [11] provides a detailed introduction.

3 TECHNICAL MATERIAL

3.1 Overview of the Stages

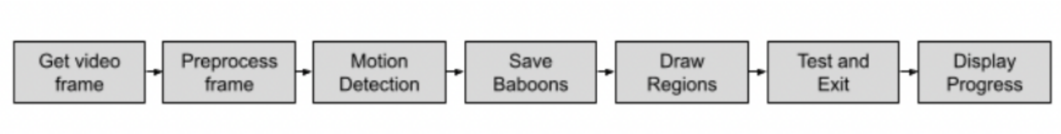


Fig. 1. Stages

The stages for processing the video footage include calls to OpenCV APIs and data structures in the C++ and CUDA implementations, with reference to the original implementation in Python [1].

(1) *Get video frame:* The input video file sequence is opened and the frames are read in a loop till the end of file.
(2) *Preprocess frame*: The frame is then pre-processed in two steps - first it is converted from RGB to gray color scale, and then blurred using the Gaussian filter.
(3) *Motion Detection*: This is the main stage for detection of the movements of the baboons. First, feature detection is applied on the pre-processed frame to obtain keypoints and their descriptors. The descriptors of current and historical (previous) frames are then matched and the best matches are passed for computing the transformation matrix for the frame. The transformation is then applied to the frame and rescaled such that it shares the origin with the historical frames. Following this, the historical frames are combined using intersections and the background is obtained from their union. This background is then subtracted from the current frame to compute the moving foreground. Noise is removed from the frame through erosion and dilation operations. Finally, the contours of the baboons are detected from the frame and bounding boxes are generated.
(4) *Save Baboons* : The coordinates of the bounding boxes generated are saved in this stage so that the testing framework can utilize it to carry out quality of results analysis.
(5) *Draw Regions*: This stage is used to embed the bounding boxes over the detected baboons in the input video. It opens up a graphical window to display the embedded video corresponding to the current frames that are being processed.
(6) *Test and Exit*: This stage checks whether the user wants to end the processing and if affirmative, it closes all the GUIs running while terminating the processing and gives an output of the benchmarking results of the different stages.
(7) *Display Progress*: The progress of the processing of the frames out of the total number of frames detected in the input video is displayed here.

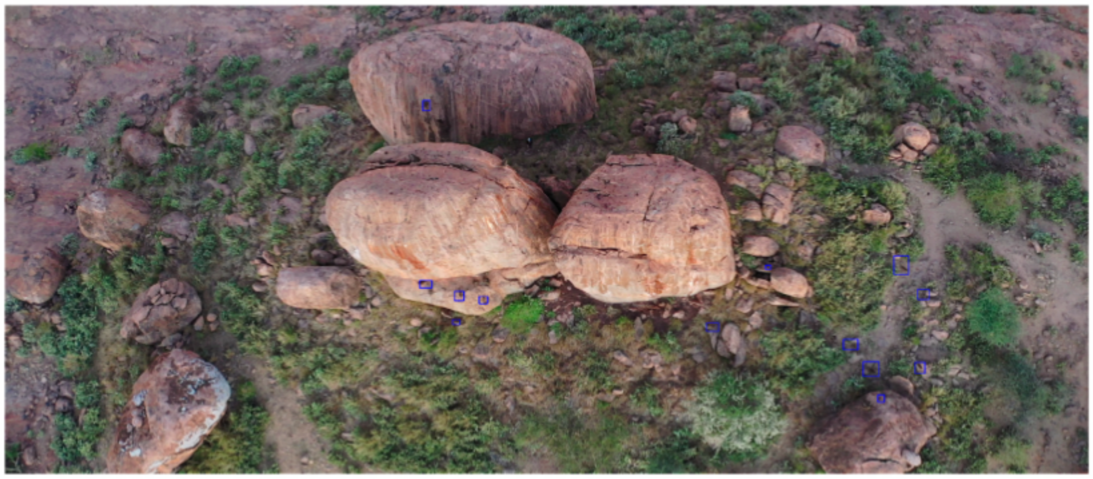


Fig. 2. Python - Frame 9 , large bounding boxes (13)

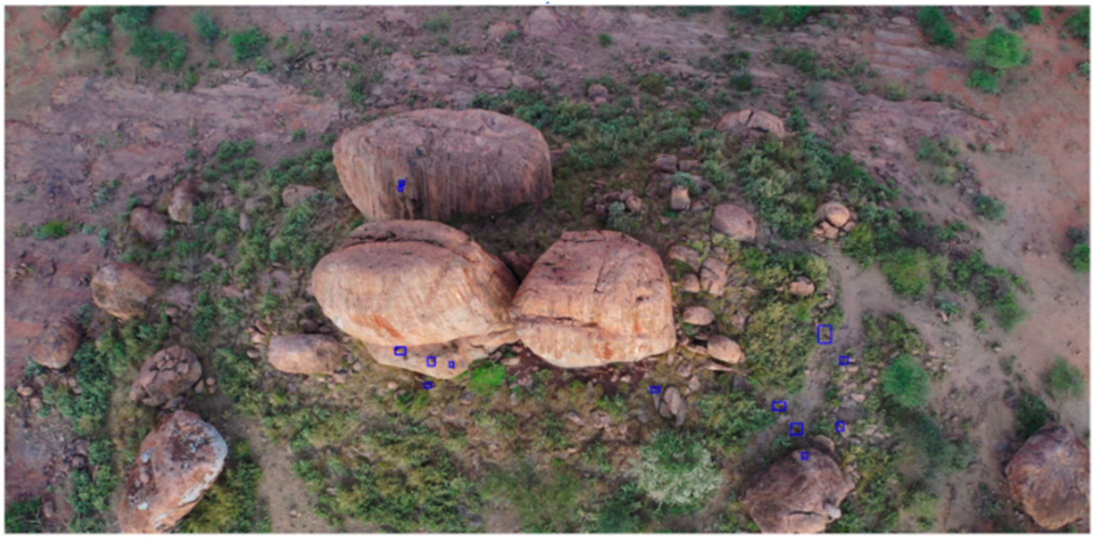


Fig. 3. C++- Frame 9 , large bounding boxes (13)

3.2 Testing

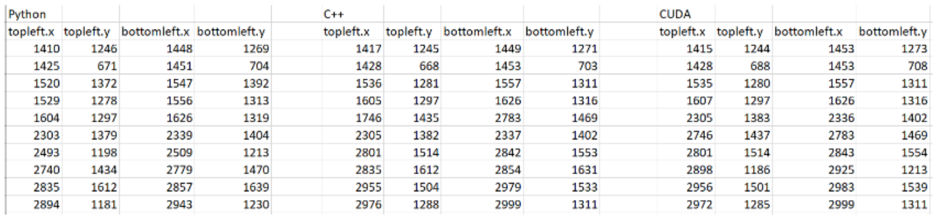
For testing the correctness of the output in the implementations, we have created a testing framework based on the library Pandas in Python. The framework takes the outputs corresponding to C++/ CUDA and the Python implementation at the end of the "Motion Detection" stage using the "Save Baboons" stage. The outputs are stored in the form of coordinates of the bounding boxes which correspond to the detected baboons in a particular frame.

The framework takes these stored coordinates as inputs and takes either the C++/CUDA or Python implementation as the reference depending on which one has detected more baboons. Since the OpenCV implementation of APIs might be different in C++/CUDA as compared to Python, C++ implementation was generating some very small bounding boxes which were not always corresponding to baboons. This was most probably due to difference in the noise filtering implementation in C++ and Python. So as a workaround, we rejected the bounding boxes which were below a certain threshold. It then goes on to find the coordinates from the other implementation which closely resembles the bounding boxes in the reference implementation. This way the framework is able to generate the metric Closely Matching Bounding Boxes (CMBBs). This gives us a good description of the quality of equality of results as a higher percentage of CMBBs indicates that both implementations are detecting similar number of baboons.

The second aspect of the framework reports the quality of the CMBB results. The framework uses the metric Intersection of Union (IOU) to compare the results from both the implementations. It takes the coordinates of the most closely resembling bounding box from outputs of both the implementations and computes the overlap and union area to determine the IOU. The framework reports the average IOU for all the CMBB between the C++/CUDA and Python implementation. An IOU higher than 60% for a frame is considered a good matching.

Table 1. Testing Results

	C++ over Python	CUDA over Python
% of CMBBs	77	76
IoU of CMBBs	72	70



Python				C++				CUDA			
topleft.x	topleft.y	bottomleft.x	bottomleft.y	topleft.x	topleft.y	bottomleft.x	bottomleft.y	topleft.x	topleft.y	bottomleft.x	bottomleft.y
1410	1246	1448	1269	1417	1245	1449	1271	1415	1244	1453	1273
1425	671	1451	704	1428	668	1453	703	1428	688	1453	708
1520	1372	1547	1392	1536	1281	1557	1311	1535	1280	1557	1311
1529	1278	1556	1313	1605	1297	1626	1316	1607	1297	1626	1316
1604	1297	1626	1319	1746	1435	2783	1469	2305	1383	2336	1402
2303	1379	2339	1404	2305	1382	2337	1402	2746	1437	2783	1469
2493	1198	2509	1213	2801	1514	2842	1553	2801	1514	2843	1554
2740	1434	2779	1470	2835	1612	2854	1631	2898	1186	2925	1213
2835	1612	2857	1639	2955	1504	2979	1533	2956	1501	2983	1539
2894	1181	2943	1230	2976	1288	2999	1311	2972	1285	2999	1311

Fig. 4. Python - Frame 9 , large bounding boxes (13)

3.3 Benchmarking

To identify areas of improvements, we decided to benchmark different sections of the implementations. As a starting point, we tried to benchmark different stages of the algorithm to do a comparison between the C++ and Python stages. To do an apple to apple comparison we had to refactor the code implementation in C++ to match the stages of python implementation.

We used standard timer calls available in C++ and Python to benchmark different stages. The benchmarking of individual stages in both Python and C++ helped us to identify the stages which were significantly slowing down the entire execution process. Additionally, we could identify the stages that were taking more than expected time in C++ and we decided to optimize them for runtime.

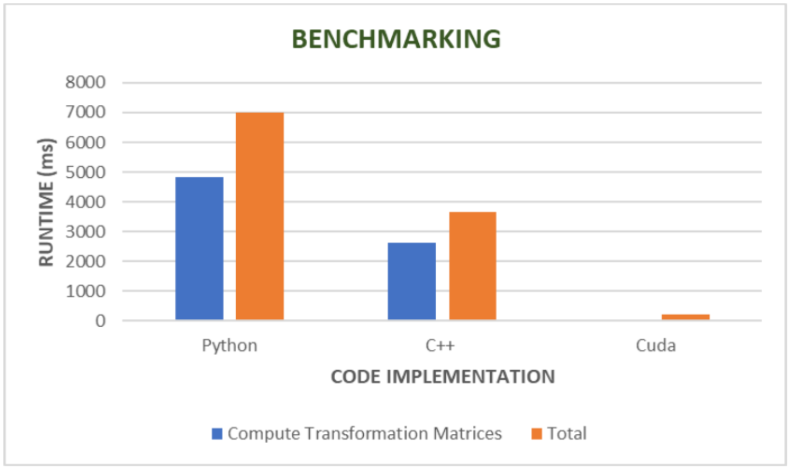


Fig. 5. Comparison of runtimes with the 3 implementations

Table 2. Benchmarking Results

	Python	C++	CUDA
Blur Gray	5.87 ms	2.83ms	0.43 ms
Compute Transformation Matrices runtime	4826 ms	2635 ms	30 ms
Total runtime	6998 ms	3657 ms	228 ms

The graph above shows the comparison of the stage that took the most runtime (Compute Transformation Matrices stage) and the total runtime of the 3 implementations. For the CUDA part, the runtime for Compute Transformation Matrices stage is very small in comparison so it's not visible on the graph.

3.4 Refactoring Existing Code for Improved Runtime Performance

We identified one stage under motion tracking, "Compute Moving Foreground", for which the Python implementation has a faster runtime than the C++ implementation. The stage is based on an algorithm proposed in [9], which uses a weight matrix and a history of dissimilarity to detect moving objectives in foreground (see Section III.C).

After carefully examining both implementations, we conjecture that the Python implementation is faster due to its usage of the NumPy library for various matrix operations which likely takes advantage of internal optimizations within the library. To improve the runtime performance of the C++ implementation, we refactored the code by making the matrix operations more efficient. Specifically, we store certain intermediate results to avoid redundant and inefficient matrix operations. Furthermore, instead of *consecutively* calling existing bitwise operations on cv::Mat objects, we manually iterate through the matrices in an efficient way [2] to perform multiple operations *simultaneously* along the way so that the runtime performance can be improved.

Further benchmarking shows that our optimization improves the C++ runtime performance on computing moving foreground from 111ms to 77ms, achieving a *1.44x* speedup.

3.5 Parallelization

For implementing the code in CUDA, we first identified those stages which took most time. Amongst that we found that in the motion detection stage, the part to find the baboons, using feature matching and homography took up around 70% of the runtime. This involves a lot of computation using matrices and hence was the best stage to implement in CUDA because the threads can compute parts of the matrices parallelly.

In our CUDA implementation, instead of the OpenCV APIs, accelerated OpenCV CUDA versions of the APIs were used. This requires the OpenCV module to be built with the CUDA flag enabled. This module provides a set of classes and functions to utilize CUDA computational capabilities. It is implemented internally using the NVIDIA CUDA runtime APIs and supports only NVIDIA GPUs.

For the primary data container in this implementation, class cv::gpu::GpuMat was used in place of cv::Mat. The interface is similar in both the classes but a major advantage provided by the former is that it keeps the data in the GPU memory. This provides an increase in performance because the data transfer for CPU to GPU memory can become a bottleneck due to the slow bandwidth on the connecting network compared to the computation throughput on the GPU. Without much change to the stages in the C++ implementation, using the CUDA versions of the APIs provided significant speedup. The stages to which these were applied are- preprocessing stage (blur the gray frame using Box filter), feature detection to obtain keypoints and their descriptors, matching of descriptors of current and historical frames, noise removal with erosion and dilation. Since we didn't have access to a GPU locally, we ran the CUDA implementation through Kubernetes on the Nautilus cluster, which has a variety of NVIDIA GPUs.

4 MILESTONES

- Milestone 1 (MVP): Port the algorithms to C++ following the CLI Chart
  - Compile a list of stages. Divide the algorithms/stages among group members
  - Create a testing framework for C++ codebase to ensure functional correctness
    Progress: *Done*
- Milestone 2 (MVP): Performance benchmarking and optimization
  - Compare the runtime of each stage between C++ and Python code bases
  - Optimize the C++ codebase to ensure minimum performance improvement
    Progress: *Done*
- Milestone 3 (Early Stretch Goal): Refactoring existing implemented code
  - Perform feasibility study on parallelizable algorithms with CUDA
    Progress: *Done*
- Milestone 4 (Long Stretch Goal): CUDA Implementation of the algorithms
  - Divide the algorithms/stages among group members
  - Testing the CUDA implementation for performance benchmarking Deliverable: Compare the runtime for the stages with the Python and C++ code
    Progress: *Done*
- *Revisions:*

  We removed this part of Milestone 4 - Feasibility study of Kalman filter [17] for multi-baboon motion tracking and prediction. In our project specification, it was considered an ultimate stretch goal, which we would attempt to complete if time permitted. It was also understood that this part of the project would be research-oriented and open-ended. After carefully examining our progress and planning for the remaining milestones, we decided to not pursue this part of the milestones, and prioritize on completing our MVP and early stretch goals.

5 CONCLUSION

We were able to complete what we had set out to achieve at the beginning of the quarter - complete the porting of all the stages from Python implementation to C++/CUDA implementation to improve the runtime of post processing of the video for detection of the baboons. Based on the Python implementation, we modified the existing implementation of the C++ code and refactored it to achieve around *2x* improvement in runtime over the Python implementation. Additionally, we ported the code to CUDA, and could achieve a significant speedup of *30x* over the Python implementation. At present, the CUDA implementation takes around 230ms to process a single frame from the input video.

Another aspect of the project was to maintain the accuracy of the outputs across the implementations; for that we were able to generate a testing framework which could help us track the correctness of the outputs across the implementations. We used Closely Matching Bounding Boxes (CMBB) and Intersection of Union (IoU) as the metrics for the quality of equality of outputs. In the final implementation, in comparison between C++/CUDA and Python, we have achieved higher than 70% average CMBB and IoU, which is considered a good match.

For future work for the project, we would like to display output with the bounding boxes for the CUDA implementation. An application to manage the GUI is needed to open up the display window for the output and we were unable to access that from the Nautilus cluster which we used to run the CUDA implementation. Additionally, we would like to explore if any other stage can be implemented in CUDA for better performance. Regarding the quality of outputs, since we have a good IOU, we can try to figure out the reasons for not having an ideal CMBB and work on implementations focusing on the noise filtering stage in C++/CUDA that will improve the CMBB further.

REFERENCES

[1] Christopher L. Crutchfield, Jake Sutton, Anh Ngo, Emmanuel Zadorian, Gabrielle Hourany, Dylan Nelson, Alvin Wang, Fiona McHenry-Crutchfield, Deborah Forster, Shirley C. Strum, Ryan Kastner, and Curt Schurgers. 2020. Baboons on the Move: Enhancing Understanding of Collective Decision Making through Automated Motion Detection from Aerial Drone Footage. (2020).

[2] OpenCV 2.4.13.7 documentation. 2022. How to scan images, lookup tables and time measurement with OpenCV. https://docs.opencv.org/2.4/doc/tutorials/core/how_to_scan_images/how_to_scan_images.html#the-efficient-way/. [accessed 06-08-2022].

[3] UCSD E4E. 2022. Baboons on the Move. https://e4e.ucsd.edu/baboons-on-the-move/. [accessed 06-08-2022].

[4] Pascal Fua and Krzysztof Lis. 2020. Comparing python, go, and c++ on the n-queens problem. *arXiv preprint arXiv:2001.02491* (2020).

[5] Lars Haalck, Michael Mangan, Barbara Webb, and Benjamin Risse. 2020. Towards image-based animal tracking in natural environments using a freely moving camera. *Journal of neuroscience methods* 330 (2020), 108455.

[6] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. 1993. Maintaining Discrete Probability Distributions Optimally. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (Lecture Notes in Computer Science, Vol. 700)*. Springer-Verlag, Berlin, 253–264.

[7] Joshua Kang and Christie Wolters. 2020. Baboons on the Move. *UCSD CSE 237D Final Report* (2020).

[8] Lutz Prechelt. 2000. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer* 33, 10 (2000), 23–29.

[9] Kumar S Ray and Soma Chakraborty. 2017. An efficient approach for object detection and tracking of objects in a video with variable background. *arXiv preprint arXiv:1706.02672* (2017).

[10] Microsoft Research. 2014. Zootracer. https://www.microsoft.com/en-us/research/project/zootracer/. [accessed 06-08-2022].

[11] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[12] Vivek Hari Sridhar, Dominique G Roche, and Simon Gingins. 2019. Tracktor: image-based automated tracking of animal movement and behaviour. *Methods in Ecology and Evolution* 10, 6 (2019), 815–820.

[13] Shirley Strum, Deborah Manzolillo Nightingale, Yvonne de Jong, and Juan Manuel Sandoval. 2008. Guess who's coming to dinner. *Swara Magazine October-December* (2008), 24–29.

[14] Sabine Stueckle and Dietmar Zinner. 2008. To follow or not to follow: decision making and leadership during the morning departure in chacma baboons. *Animal Behaviour* 75, 6 (2008), 1995–2004.

[15] Tristan Walter and Iain D Couzin. 2021. TRex, a fast multi-animal tracking system with markerless identification, and 2D estimation of posture and visual fields. *Elife* 10 (2021), e64000.

[16] Sherwood Larned Washburn and Irven DeVore. 1961. The social life of baboons. *Scientific American* 204, 6 (1961), 62–71.

[17] Greg Welch, Gary Bishop, et al. 1995. An introduction to the Kalman filter. (1995).

[18] Farzeen Zehra, Maha Javed, Darakhshan Khan, and Maria Pasha. 2020. Comparative analysis of C++ and Python in terms of memory and time. (2020).