Real-Time Stereo Vision for Collision Detection on Autonomous UAVs

Aleksandar Stanoev, Nicolas Audinet, Scott Tancock, Naim Dahnoun
Emails: as14622@my.bristol.ac.uk, na14285@my.bristol.ac.uk, st12660@my.bristol.ac.uk, naim.dahnoun@bristol.ac.uk
Department of Electrical and Electronic Engineering
University of Bristol
Bristol, U.K.

Abstract—**Collision detection is an important unsolved problem in the domain of modern UAV, which would enable safe navigation in unknown environments. Stereo vision provides a compact, lightweight and low-power solution. This paper describes an adaptive system for achieving real-time stereo vision for collision detection on an embedded GPU. Several optimisations are described including using sensor fusion with an ultrasonic sensor to better filter noise, organising the computations to take advantage of the platform's heterogeneous architecture and using GPU textures to benefit from caching. A discussion of the hardware features is provided, followed by the algorithm and implementation details for disparity calculations and finally a method for identifying objects from a disparity map. The system was implemented on an NVIDIA Tegra X1, achieving 48 FPS on a 320x240 image.**

I. Introduction

UAV[1] technology has seen a lot of interest in many industries in recent years. Applications range from military [1], monitoring, delivery, surveying, first respondents in accidents and even delivering vaccines to remote locations [2][3]. In many of these applications, UAVs require the ability to autonomously navigate through an unknown environment without crashing into obstacles.

Stereo vision is a promising avenue to approach the problem of detecting obstacles. Stereo vision relies on the differences between two images of the same scene captured at the same time from two cameras at a known offset. Cameras are cheap, light and low-powered, and computation power is becoming increasingly so. Additionally, cameras capture a wide scene almost instantaneously, contrasting with other methods such as LiDAR or infrared time-of-flight depth sensors which require forced IR illumination.

Stereo vision depth maps also provide a dense set of depth measurements, which can be used to accurately map the environment. This map can then be used to recommend a path to the UAV navigation system to avoid the detected objects [4]. Although stereo vision technology has been an established area of research for many years, its use in real-time collision detection still remains an unsolved problem. A variety of different approaches, such as [4], [5], and [6] are common. An important trade-off to always consider in a stereo vision system is speed versus quality of the disparity map. Recently, a new generation of embedded GPUs from NVIDIA - the Maxwell-based Tegra X1, has made substantial progress in the processing power available for embedded applications, allowing for fast real-time stereo vision systems with a relatively low power profile [7].

This paper focuses on describing and demonstrating a collision detection pipeline targeted at a modern embedded GPU architecture. Section II contains an overview of the hardware used for this implementation. Section III contains an overview of the complete collision detection system. In Section IV, the theory and the implementation of real-time disparity map generation from a stereo image pair is described. Section V deals with the filtering techniques applied to the disparity map for noise reduction. Section VI illustrates a method for identifying objects from the disparity map. Section VII contains benchmark results from the practical implementation of the proposed method. Finally, Section VIII summarises the achievements and results of the paper, and suggests future work that could be performed.

II. Hardware considerations

The UAV application that the proposed system targets necessitates a compact and power-efficient platform. Due to the highly parallel nature of the chosen disparity algorithm (explained in section IV-A), it can be efficiently offloaded to compute accelerators such as graphic processors, digital signal processors or other massively parallel systems.

A. The Tegra X1 SoC

The proposed GPGPU[2] implementation of this algorithm uses NVIDIA CUDA technology and targets the NVIDIA Jetson TX1 compute module. This is a platform using the NVIDIA Tegra X1 mobile processor, which contains four 64-bit ARM A57 CPU cores and an NVIDIA Maxwell GPU with 256 CUDA cores (Fig. 1). The Tegra X1 SoC[3] is oriented towards UAV/automotive computer vision and autopilot applications as well as neural networks.

The Jetson TX1 runs Linux, has a memory bandwidth of 25.6GB/s and its GPU can perform up to 1 TeraFLOP of 16-bit

[1]Unmanned aerial vehicle

[2]General-purpose computing on graphics processing units

[3]System-on-Chip

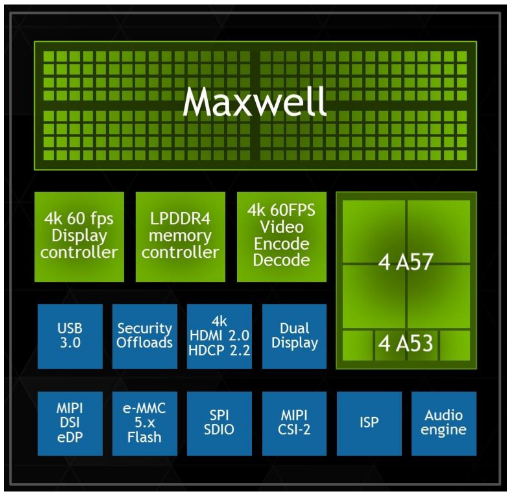

Fig. 1: NVIDIA Tegra X1 mobile SoC architecture [7]

floating point calculations with a maximum thermal design power of 15 watts [8][7].

The main motivation for choosing this platform was the high relative performance among other ARM-based solutions coupled with a familiar (CUDA-capable) GPU, while consuming a fraction of the power of an entire drone platform. The combination of a standard Linux distribution and CUDA API allowed for rapid development on ordinary x86/NVIDIA GeForce desktop hardware with no changes required when running on the target device.

B. *The Maxwell GPU architecture*

Massively parallel platforms require different programming paradigms compared to the usual sequential model. The approach taken by CUDA and OpenCL is modelled around the physical architecture of graphics processors. It involves launching kernels - blocks of code which perform a single iteration of a computation. These are launched by the CUDA runtime multiple times so as to best occupy the underlying hardware. The device is controlled by a host-side program which is responsible for enqueuing kernels and copying memory on and off the GPU, while also being able to perform computation on the CPU.

The Maxwell architecture consists of Graphics Processing Clusters, which in turn contain a number of Streaming Multiprocessors (SMM). Each streaming multiprocessor contains CUDA Cores, organised in warps. Each warp runs 32 parallel threads which execute instructions in lockstep.

The kernels run on the CUDA cores in parallel and make use of the register file and texture cache to efficiently store data without hitting global memory, which has a much higher access latency.

Additional Maxwell architecture details are available in NVIDIA's GTX 980 whitepaper [9] and a general warp scheduling and dispatch overview is provided in NVIDIA's Kepler GK110 Architecture whitepaper [10].

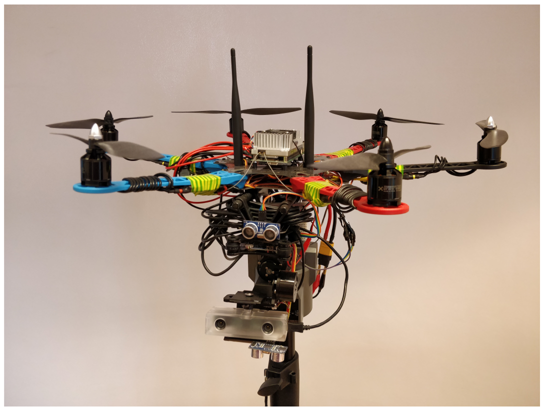

Fig. 2: S550 hexacopter

III. Collision detection system overview

The UAV control system consists of the Jetson TX1 module mounted on an Auvidea J120 carrier board [11] and a Tara stereo vision camera [12].

Flight control inputs are accepted from various sources by an application which takes into account the output of the proposed collision detection system along with input from various other sensors. These inputs are then converted into Yaw/Pitch/Roll/Throttle values which are relayed to a flight controller, performing PID control to keep the aerial platform stable. The proof of concept has been implemented on an S550 hexacopter [13] (Fig. 2) utilising the Naze32 flight control platform [14], however no autonomous tests have been performed yet.

The e-con Systems Tara stereo vision camera has been chosen due to its synchronised 60fps 10-bit uncompressed grayscale output and out-of-the box calibration. The manufacturer provides a Software Development Kit which wraps communication with the camera and image rectification in an interface accessible from C++ OpenCV.

Camera images are acquired by the ARM host-side software (section IV-D) over a USB 3.0 bus. Each frame is then rectified to compensate for camera lens distortion before being copied to the GPU. The GPU performs disparity calculations (detailed in section IV-B) and executes the guided filter (section V-A) and morphological filters (section V-B). Post-processing on the previous frame is executed in parallel on the CPU while the GPU kernels are running to maximise throughput (Fig. 3).

IV. Disparity implementation

There are many approaches to calculating the disparity of stereo images. They can be classified into three categories: global, semi-global and local methods.

Global methods involve minimising an energy function over all disparity values. These methods tend to output good results, but also have a high complexity and are not suitable for real-time embedded processing. Semi-global methods share similar characteristics with global methods [15].

Local methods make the assumption that all the information needed to calculate the disparity of a given pixel can be found in its local neighbourhood. Therefore, these methods tend to

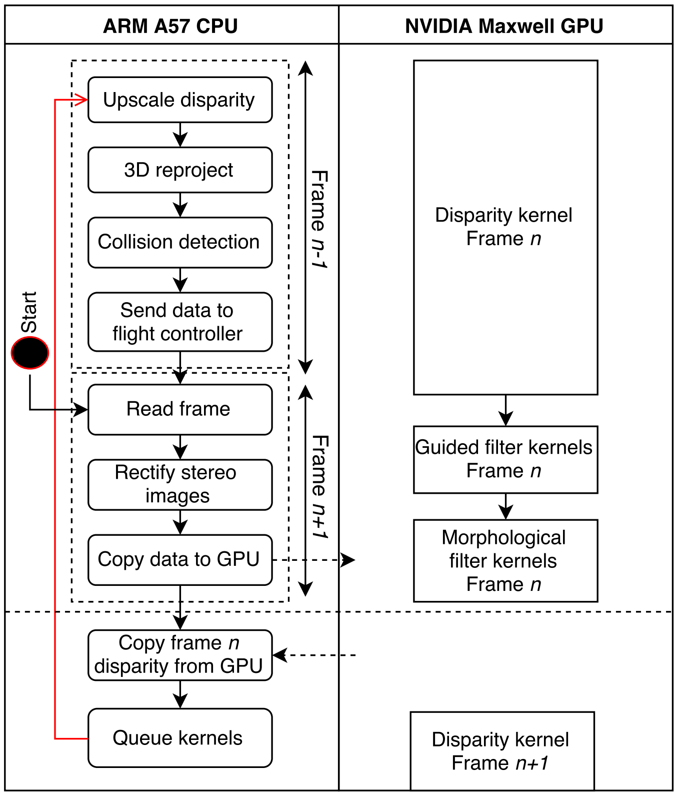
ARM A57 CPU
NVIDIA Maxwell GPU
Start
Upscale disparity
3D reproject
Collision detection
Send data to flight controller
Read frame
Rectify stereo images
Copy data to GPU
Copy frame *n* disparity from GPU
Queue kernels
Frame *n-1*
Frame *n+1*
Disparity kernel Frame *n*
Guided filter kernels Frame *n*
Morphological filter kernels Frame *n*
Disparity kernel Frame *n+1*

Fig. 3: Pipelined algorithm implementation

have a lower complexity and faster execution time than global methods, and were chosen for our real-time implementation. However, they also tend to be significantly noisier than their global counterparts.

A. Algorithm

As defined by Scharstein and Szeliski [16], local stereo vision methods can be defined in four steps: cost computation, cost aggregation, disparity selection and disparity refinement.

In cost computation, the pixels in the support window around the two pixels to be matched are compared. In the cost aggregation step, these comparisons are used to produce a likelihood of the two pixels matching. A simple and fast way of achieving this is calculating the Sum of Absolute Differences (SAD),

$$SAD\left(x,y,d\right)=\sum_{(i,j)\in W}\left|I_L\left(i,j\right)-I_R\left(i-d,j\right)\right| \quad (1)$$

where x and y are the coordinates of the pixel in question, d is the disparity to be tested, W is the set of all pixels in the support window centred at (x,y) and $I_L(i,j)$ and $I_R(i,j)$ are the intensity values at coordinates (i,j) of the left and right input image respectively. A low SAD score between two regions indicates that the difference between them is small, and therefore that the two pixels likely match. This is the method used in our disparity calculations.

Various other matching methods using support windows exist. The Sum of Squared Differences [17] method squares the pixel difference instead of finding the absolute value, and is more sensitive to outlier noise. Normalised Cross Correlation (NCC) [18] seeks to statistically compensate for differences in gain and bias, but also blurs discontinuities (edges). Finally the Census Transform (CT) [19] finds the Hamming difference between two bit strings created by comparing all the pixels in the support window to the centre pixel. This method has good outlier tolerance, but performs poorly in regions with repeated structures. Although NCC and CT methods produce less noisy disparity maps, they require significantly more computing power. Therefore, it was chosen to use the faster SAD comparisons and filter the depth map at a later stage.

To select the best disparity, a Winner Takes All (WTA) approach was used. This method selects the disparity with the lowest SAD score as the disparity from the available disparity range. Finally, two different filters are applied to the resulting disparity map to reduce noise (see Section V).

B. GPU disparity implementation

The implemented algorithm performs left-right stereo matching and filtering on the GPU. Such disparity calculations are inherently memory-bound and can exploit the massive parallelism of the hardware due to individual pixel calculations being independent of each other. The matching is performed by taking a square window of pixels from both images and correlating them by calculating the SAD. Due to the nature of the square sliding window, the run time complexity is $O(d^2nr)$ where d is the side length of the square window, r is the maximum disparity range and n is the total number of pixels in the image. Each window in the left image (which is not shifted by the disparity value being considered) is calculated in parallel with the others using the parallelism available in the GPU.

The left window is stored fully in GPU registers - this is achieved by keeping it in a statically indexed array such that all accesses can be resolved at compile time. For every window in the left image, matches are performed by sliding a window horizontally on the right camera image up to the search range. This implies that there is a large overlap between every adjacent window, as each next one is shifted to the left by a single column of pixels.

These factors provide an ideal use case for textures, which allow for both efficient random access to arrays as well as a 2D aware cache. By mapping the stereo images to 2D CUDA textures, recently accessed texels[4] would be cached in the texture cache, thus avoiding reloading the entire window from global memory.

Development was done on an NVIDIA GeForce GTX 770 (Kepler) GPU with CUDA 8.0. The achieved occupation of the development hardware is 48.5% which is enough to partially hide the effect of memory latency. This occupation value is caused by the high use of registers (used for window caching) per thread.

Upon profiling on the GTX 770, the software exhibits a texture cache read rate of 301.8GB/s but is bound by global memory access speed and latency, peaking at 117.9GB/s. This

[4]Texture elements

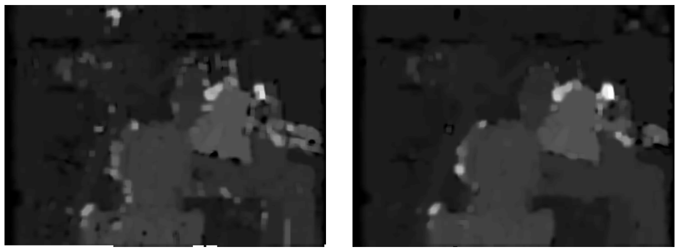
(a) 7x7 window size (b) 11x11 window size

Fig. 4: Comparison of disparity quality for varying window sizes (Tsukuba image pairs)

shows that compute optimisations won't have a measurable effect on the speed of the kernel and instead speedups can be achieved by either increasing the memory bandwidth (by using more powerful and thus more power-hungry hardware, which defeats the embedded focus of this paper) or by decreasing the amount of pixels processed by the kernel (a product of the window size).

C. *Performance impact of window sizes*

Increasing the support window size results in a loss of details but decreases the number of outliers and noise (Fig. 4).

As noted in the previous section, the window size needs to be constant to allow for loop unrolling by the GPU compiler. A way to still allow for varying window sizes at runtime is to build several versions of the disparity kernel, each with different window dimensions and enqueue whichever window size is required.

In UAV applications, this allows for varying the accuracy of the collision detection subsystem with regards to the flight speed. Faster speeds require lower latencies to avoid collision, but those are obtained by sacrificing accuracy which can be regained if a longer frame time is acceptable.

A combination of ultrasonic [20] and laser [21] distance sensors allows for a fail-safe to be implemented alongside the proposed computer vision approach. While these sensors do not allow for obstacle avoidance, they do allow for switching between the two disparity accuracy modes - fast (46 fps) but noisy and slower (20 fps) but with less noise without risking collision. Since those sensors have a wide field of view (25 degrees for the VL53L0X laser [21] and 15 degrees for the ultrasonic sensor [20]), their output can also be used as an adaptive threshold for filtering out erroneous matches in the resulting disparity. The noise in disparities calculated with a smaller window is usually caused by occlusions or untextured surfaces and manifests itself as an outlier area of high disparity. If the distance of the nearest object reported by the time-of-flight sensors is larger than this erroneous detection, these outliers can be culled safely.

The achieved performance with varying window parameters is detailed in section VII.

D. *Host-side code*

The host code (implemented in C++) is tasked with acquiring frames from the stereo camera, copying memory content to and from the GPU and performing postprocessing on each calculated disparity. This is then passed over to the collision detection section of this system (section VI) which performs 3D reprojection and sends collision data to the flight controller.

By refactoring the code such that there are always two frames in flight (Fig. 3), the implementation can benefit from asynchronous kernel launches and perform computation while the GPU is busy, instead of blocking. This approach considerably increases the program's throughput, but requires careful balancing of the workloads to prevent one side (CPU or GPU) from becoming a large bottleneck. Camera frame capture was implemented in a separate thread to prevent reliance on the camera frame rate and any possible frame time jitter due to capture delay.

V. Filtering

SAD region-based disparity calculation is fast but noisy. This reduces the accuracy of the collision avoidance system, increasing the error of the distance measurements and potentially introducing artifacts in the image. To counteract this, the disparity map is filtered to improve the Signal-to-Noise Ratio. Two types of filters were used: a guided filter and an opening morphological transformation.

A. *Guided Filtering*

For this application, it was chosen to use a guided filter [22], which performs edge-aware blurring (averaging) of the image. This filter aims to average the areas of the image with low variance (objects) whilst keeping areas of high variance (edges) intact. It does this by using a guidance image, and assuming that the output is a linear transformation of the guidance image. This would therefore maintain the edge structure in the guidance image.

Additionally, the calculations for each pixel are independent of each other, making the algorithm highly parallelisable and a good fit for running on the graphics processor.

The filter is implemented on the graphics processor as a two-stage process. The first pass computes A and B values by sliding a window over the complete disparity image and performing the calculations detailed in section IV. This two-step process is necessary as the final step of the filter averages the values of A and B over a window for every pixel. This creates a data dependency - where every pixel depends on the computed values of those surrounding it. The first pass computes those variables for every pixel independently and populates a pair of buffers which are subsequently used by the second pass to compute the filtered value for every pixel.

B. *Morphological filtering*

When taking a stereo image, it is likely that an object obstructs the view of one of the cameras. Therefore, when trying to match pixels in the occluded region, the algorithm will fail and usually return unpredictable results. Some of these will be attenuated by the guided filter, but often this still leads to small bright regions in the disparity map.

To counteract this problem, an opening morphological transformation is applied to the disparity map. An opening transformation is composed of two steps: an erosion transformation followed by a dilation transformation. The erosion transformation assigns to every pixel in the image the minimum value in a local region defined by a kernel (or structuring element). Symmetrically, the dilation transformation assigns to every pixel in the image the maximum value in the kernel.

When a dilation transformation is performed after an erosion transformation with the same kernel, it will restore the bright regions to their original size, albeit altering the value of the pixels.

Similarly to the guided filter, the opening transformation is highly parallelisable since the value assigned to each pixel is independent from the value assigned to all other pixels. This calculation is therefore also executed on the GPU.

The implementation of the erosion and dilation kernels is similar to that of the second stage of the guided filter. An aggregate min operation for erosion and max operation for dilation is performed on a window with a configurable size.

The combined opening transformation is performed in-place on the GPU disparity array by utilising an intermediate buffer such that erosion reads from the main buffer and writes into the temporary buffer and dilation reads from the temporary buffer and writes back into the main buffer.

VI. Collision detection

Once the disparity has been calculated for each pixel in the input stereo image, it has to be used to recognise objects. The first step is to convert the disparities to depth measurements. The depth measurements are then used to identify prominent objects that need to be avoided.

A. Conversion to depth map

The disparity map only contains the difference in pixel positions between the same pixel in the left and right stereo image (the disparity). This distance is directly linked to the distance between the object and the stereo camera. To retrieve the depth information from the disparity map, one can use the following formula:

$$Z = \frac{fT}{d}$$

where Z is the perpendicular distance from the image plane to the object, T is the distance between the two focal points (the baseline), f is the distance between the focal point and the image plane (the focal length), and d is the disparity. It is possible to use this equation, rather than the more complex epipolar geometry, as the cameras had been physically calibrated to avoid any alignment issues.

In practice, two OpenCV functions were used to achieve this. The $stereoRectify()$ function is run by the Tara camera Software Development Kit, and returns the Q matrix which contains the baseline and focal length parameters. This matrix is then used by the $reprojectImageTo3D()$ function that takes the disparity and returns an array of three images containing the 3D coordinates of each pixel. The final depth map is then obtained by taking the Euclidean distance from the origin of the points.

B. Object Identification

The next task of the system is to recognise objects to avoid from the depth map. This is achieved in three stages:

1) threshold the depth map
2) label the various different objects
3) model the objects with spheres

The aim of the thresholding step is to simplify the depth map into a binary image (see 5). The white pixels in this binary image will be the obstacle pixels, or pixels that are closer than a certain threshold to the drone. Dark pixels will be pixels with a depth value beyond the threshold, which are ignored. The threshold that determines how far ahead to look is adaptively decided based on the speed of the drone. If the drone is travelling fast, it will take longer to stop and avoid obstacles, and therefore the threshold will be large. Conversely, when the drone is travelling slowly, there is no need to consider objects far away from the UAV and therefore the threshold can be smaller.

Once a binary image is obtained from thresholding the depth map, the Connected Component Labelling (CCL) algorithm is executed (see 5). This algorithm outputs an image where all white connected pixel clusters are labelled [23], [24]. Dark pixels are ignored and labelled as background. The purpose of this step is to identify which pixels belong to the same object that needs to be avoided. Note that it is assumed that disconnected pixels do not belong to the same object.

Each relevant pixel is labelled, then the image is scanned again to identify more information about the detected objects. Every detected object is modelled as a sphere, based on the model described in [4]. The centre of the sphere is calculated as the mean of the 2D image coordinates of the pixels within each object. The radius of the sphere is calculated by iterating over all the pixels in the object again and finding the pixel that is furthest away from the centre (see Fig. 5). Finally, using the depth information, the 3D coordinates of the centre of the sphere relative to the UAV and it's radius in meters are approximated by averaging, similarly to the 2D centre and radius.

VII. Benchmarks

The implemented disparity algorithm performs at 48 fps, using a 7x7 window, (faster kernel) on a 320x240 8-bit single channel stereo image with an average frame time of 20 ms (over 500 frames). Increasing the window size to 11x11 (slower kernel) results in a frame time of 47 ms and a frame rate of 21 fps.

The Jetson TX1 exhibited an idle power draw of 4.9 watts and drew 13.5 watts while performing video capture (USB bus powered camera) and disparity calculations using the 7x7 window implementation.

The system has not been comprehensively tested yet due to time constraints. Preliminary tests have been performed by moving objects towards and away from the hexacopter. The system successfully identified when the objects moved dangerously close to the drone and provided alternative aiming points.

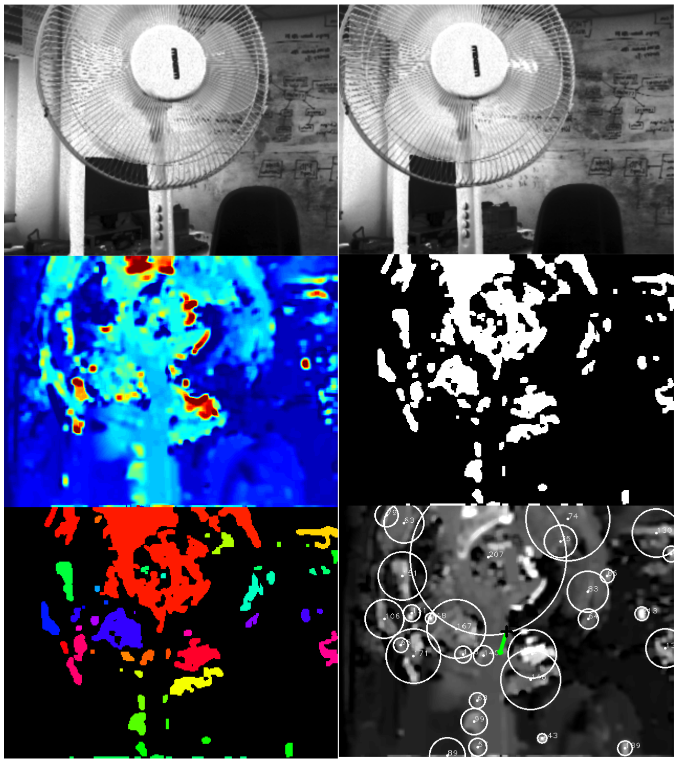

Fig. 5: Example of collision detection on a single object. Displays the original images (top), the disparity map (middle left), the thresholded disparity map (middle right), the output of CCL (bottom left) and the output of the collision avoidance (bottom right). Taken while the hexacopter was placed on a desk.

VIII. Conclusion

In conclusion, this paper demonstrated a stereo vision collision avoidance system for UAVs implemented on an NVIDIA Tegra X1 embedded GPU. It discussed the different features of a suitable embedded GPU for autonomous UAV image processing. SAD disparity was chosen due to its speed and simplicity, and implemented on a GPU using CUDA. Noise reduction operations were executed on the resulting disparity map, including using a guided filter, an opening transformation and sensor data fusion. The filtered disparity map was converted to a depth map, adaptively thresholded and analysed to identify potentially dangerous objects in the path of the UAV. Several optimisations to the systems were used, such as a method of reorganising calculations to take full advantage of the heterogeneous architecture, and efficiently using the GPU texture cache in disparity calculations.

The system ran at a maximum of 48 FPS on a 320x240 8-bit single channel stereo image and a power consumption of 13.5 Watts when performing video capture and disparity calculations. Furthermore, two disparity kernels with different window sizes were compiled and used adaptively to change the speed or quality of the disparity computation based on the speed of the UAV.

References

[1] D. Gregory, "From a view to a kill: Drones and late modern war," *Theory, Culture & Society*, vol. 28, no. 7-8, pp. 188–215, 2011.

[2] J. Javelosa. (2016, August) Drone-delivered medical supplies set to bring vaccines to remote areas. [Online]. Available: https://futurism.com/drone-delivered-medical-supplies-set-to-bring-vaccines-to-remote-areas/

[3] N. Wingfield. (2016, August) A field guide to civilian drones. [Online]. Available: https://www.nytimes.com/interactive/2015/technology/guide-to-civilian-drones.html

[4] J. Park and Y. Kim, "Collision avoidance for quadrotor using stereo vision depth maps," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 51, no. 4, pp. 3226–3241, 2015.

[5] S. Hrabar, "3d path planning and stereo-based obstacle avoidance for rotorcraft uavs," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 807–814.

[6] S. Hrabar, G. S. Sukhatme, P. Corke, K. Usher, and J. Roberts, "Combined optic-flow and stereo-based navigation of urban canyons for a uav," in *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. IEEE, 2005, pp. 3309–3316.

[7] NVIDIA, "NVIDIA Tegra X1: NVIDIA's new mobile superchip," Tech. Rep., January 2015. [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[8] D. Franklin, "NVIDIA Jetson TX1 supercomputer-on-module drives next wave of autonomous machines," https://devblogs.nvidia.com/parallelforall/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/, 2015, accessed: 2017-04-26.

[9] NVIDIA, "NVIDIA GeForce GTX 980 Whitepaper," Tech. Rep., 2014. [Online]. Available: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[10] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," Tech. Rep., 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[11] "J120 carrier board for the NVIDIA Jetson TX1," https://auvidea.com/j120/, accessed: 2017-04-26.

[12] "Tara - USB 3.0 stereo vision camera," https://www.e-consystems.com/3D-USB-stereo-camera.asp, accessed: 2017-04-26.

[13] "HobbyKing S550 Hexcopter Combo," https://hobbyking.com/en_us/hobbykingtm-s550-hexcopter-combo-frame-esc-s-and-motors-arf.html, accessed: 2017-09-04.

[14] "Naze 32 full 10 DOF flight controller rev 6," https://www.unmannedtechshop.co.uk/naze-32-full-10-dof-flight-controller-rev-6/, accessed: 2017-04-26.

[15] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[16] D. Scharstein, R. Szeliski, and R. Zabih, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," in *Stereo and Multi-Baseline Vision, 2001.(SMBV 2001). Proceedings. IEEE Workshop on*. IEEE, 2001, pp. 131–140.

[17] R. Yang and M. Pollefeys, "Multi-resolution real-time stereo on commodity graphics hardware," in *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, vol. 1. IEEE, 2003, pp. I–I.

[18] H. Hirschmuller and D. Scharstein, "Evaluation of cost functions for stereo matching," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.

[19] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze, "A fast stereo matching algorithm suitable for embedded real-time systems," *Computer Vision and Image Understanding*, vol. 114, no. 11, pp. 1180–1202, 2010.

[20] "Ultrasonic ranging module HC - SR04," https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf, accessed: 2017-09-04.

[21] STMicroelectronics, "VL53L0X - World smallest Time-of-Flight ranging and gesture detection sensor," Tech. Rep., 2016.

[22] K. He, J. Sun, and X. Tang, "Guided image filtering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 6, pp. 1397–1409, 2013.

[23] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.

[24] L. He, Y. Chao, and K. Suzuki, "A run-based two-scan labeling algorithm," *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 749–756, 2008.