# GPU Accelerated Non-Parametric Background Detection

**William Porr, Alireza Tavakkoli**

**Abstract** Accurate background subtraction an essential tool for high level computer vision applications. However as research continues to increase the accuracy of background subtraction algorithms, computational efficiency has often suffered as a result of increased complexity. Consequentially, many sophisticated algorithms are unable to maintain real-time speeds with increasingly high resolution video inputs. To combat this unfortunate reality, we propose to exploit the inherently parallelizable nature of background subtraction algorithms by making use of NVIDIA's parallel computing platform known as CUDA. By using the CUDA interface to execute parallel tasks in the Graphics Processing Unit (GPU), we are able to achieve up to a 8x speed up using the background subtraction implementation proposed in (Porr, 2017).

## 1 Introduction

Accurate separation of foreground from background is essential for many computer vision and video surveillance applications. The separation serves as a basis from which other, higher level applications can be conducted more efficiently. Such higher-level applications include tasks such as object tracking or, as what was done in (Elgammal, Duraiswami, Harwood, & Davis, 2002), the segmentation of body parts. The general approach is to create a statistical model of the background at either each pixel or a larger surrounding region using the known values from the video feed. Then for each incoming frame, the pixel or region models are referenced using the incoming values. This "referencing" will then return a value which reflects how closely the incoming value corresponds to the background model. Those values with low probabilities of belonging to the background model are to be designated as foreground.

There are more simplistic approaches to background subtraction which result in incredibly fast execution times such as (Manzanera, 2007), yet these methods often yield segmentation results which cannot be relied upon for higher level applications. On the other hand, there are extremely accurate algorithms, such as (Maddalena & Petrosino, 2008), that are extremely complex and consequentially suffer from high computational load. One method researchers have used to accelerate the speed of these more sophisticated algorithms is to push some portion of the computation to the GPU. The GPU is designed to handle massively parallel tasks, making it a useful tool in background subtraction processing since each pixel location in a video feed is often able to be processed independently.

In this paper we will explain the methods we used to integrate GPU processing into the background subtraction implementation proposed in (Porr, 2017). This scheme develops a pixel-wise non parametric probability density function. This model is then updated by applying dynamic learning weights to observed values in the density function. The rest of the paper is outlined as follows. Section 2 will explain the theory behind the implementation used. Section 3 will explain in detail the construction of the GPU and the CUDA interface. Section 4 will explain the methods used to implement the GPU processing and the strategies used to maximize efficiency. And finally we will have our experimental results in section 5 followed by a conclusion.

## 2 Non-Parametric Background Detection

The scheme used in (Porr, 2017) creates pixel-wise non-parametric density functions using observed values from the various frames of a video feed. Each function represents the probability density function for one channel of one pixel in the video frame, and this model is updated as new intensities are observed at the pixel location. A model for one channel of location $\mathbf{x}$ can be represented at time $t$ by:

$$\overset{\sim t}{\theta}(\mathbf{x}) = \alpha \overset{\sim t-1}{\theta}(\mathbf{x}^t) - \beta \overset{\sim t-1}{\theta}(\sim \mathbf{x}^t) \tag{1}$$

such that

$$\sum_{i=0}^{D} \overset{\sim t}{\theta}(\mathbf{x}_i) = 1 \tag{2}$$

where $\mathbf{x^t}$ is the most recently observed channel intensity at location $\mathbf{x}$, $\sim \mathbf{x^t}$ reflects all values in the function that are not $\mathbf{x^t}$, and $D$ is the size of the domain of the function. $\alpha$ is the value by which $\overset{\sim t-1}{\theta}(\mathbf{x^t})$ is updated at time $t$, and $\beta$ is the value by which all unobserved values of the function are forgotten. In this sense, $\alpha$ reflects the learning rate of the function, and $\beta$ the forgetting rate. For our purposes, $\alpha$ and $\beta$ can be substituted by linear functions, such that:

$$\alpha = \frac{N^{t-1} - (\overset{\sim t-1}{\theta}(\mathbf{x}^t) \cdot N^{t-1})}{(N^{t-1})^2 + N^{t-1}} \tag{3}$$

and

$$\beta = \frac{\overset{\sim t-1}{\theta}(\mathbf{x}^t) \cdot N^{t-1}}{(N^{t-1})^2 + N^{t-1}} \tag{4}$$

where $N^t$ is the number of samples in the function with relation to time $t$. These functions reflect decreasing values of $\alpha$ and $\beta$ which approach 0 as $N \rightarrow \infty$. We chose these dynamic values of $\alpha$ and $\beta$ for the computational efficiency and accuracy it provides. A decreasing learning and forgetting rate allows for the formation of a general model quickly in the beginning of the video sequence since earlier frames will have the largest influence. This rough model is then finely tuned during latter frames of the sequence using the lowered rates.

One may desire for a learning rate which approaches a constant value rather than zero as $N \rightarrow \infty$. In that case, the eq.1 could be reworked as:

$$\overset{\sim t}{\theta}(\mathbf{x}) = \frac{\alpha \overset{\sim t-1}{\theta}(\mathbf{x^t})}{1+\alpha} + \frac{\overset{\sim t-1}{\theta}(\sim \mathbf{x^t})}{1+\alpha} \tag{5}$$

where

$$\alpha = \frac{1 + Z \cdot N^{t-1}}{N^{t-1}} \tag{6}$$

where $Z$ is the constant $\alpha$ will approach as $N \rightarrow \infty$. This approach would have the benefit of increased adaptability at later frames.

In order to determine if location $\mathbf{x}$ is a background pixel, the process used in eq.1 to model the function at time $t$ is then repeated for the other two channels of pixel $\mathbf{x}$. In order to calculate the probability of $\mathbf{x}$ being a part of the background using the three probabilistic values gathered from eq.1, we use the following equation:

$$P(BG|\mathbf{x}) = \overset{\sim t}{\theta}(\mathbf{x}_B^t) \cdot \overset{\sim t}{\theta}(\mathbf{x}_G^t) \cdot \overset{\sim t}{\theta}(\mathbf{x}_R^t) \tag{7}$$

By multiplying these probabilities, we are able to acquire a probabilistic value which assumes a statistical independence at each channel. To decide which values are associated with the background and which are to be considered foreground, we establish a global threshold value $th$. Now, we determine that pixel $\mathbf{x}$ is a background pixel only if its probability as gathered from eq.7 is greater than $th$. Also, for the purposes of computational efficiency and improving end results, we adopted a parameter $\sigma$ to serve as a bandwidth for the probability density function. This bandwidth should be chosen in such a way that the effect of random noise in the video sequence is diminished, while avoiding the effects of over-smoothing.

## 3 Graphics Processing Unit

In an attempt to satiate the market demand for high quality consumer and professional graphics, GPUs have developed into a highly parallel many-core processing unit with high memory bandwidth. In order to leverage this device with such immense parallel computing capabilities, NVIDIA introduced CUDA as a general purpose parallel computing platform which comes with a built in software environment, effectively integrating GPU instruction capabilities into popular programming languages such as C (“Cuda C Programming Guide”, 2017). CUDA allows developers to create and call CUDA functions, known as *kernels*, which are individually executed on the GPU by a specified number of threads in parallel. The number of threads specified is expressed as a number of uniform blocks within a grid, each block containing a chosen number of threads, and a two-dimensional grid containing a number of blocks (“Cuda C Programming Guide”, 2017).

The NVIDIA architecture is built around an array of multiprocessors. When a kernel grid is called by a host program, all the threads of an individual block are executed concurrently on one multiprocessor, and each multiprocessor is able to execute one or more blocks depending on the availability of computational resources. In order to manage hundreds of concurrently executing processing threads, NVIDIA incorporates Single-Instruction, Multiple-Thread architecture (SIMT). This architecture manages and instructs groups of 32 threads known as *warps* (“Cuda C Programming Guide”, 2017).

The two major pools of memory on a device are the global and shared memory spaces. Global memory is the larger pool accessible by all threads of the device. In contrast, shared memory is only accessible by all the threads within a single block. The reason this distinction is important is because global memory access is much slower than shared memory access, prompting a developer to replace global memory access with shared memory access wherever possible ("Cuda C Programming Guide", 2017).

## 4 GPU Implementation

As was stated previously, background subtraction inherently possesses an extremely parallelizable processing structure. A simple and popular method commonly used, such as in (Liu, 2015), (Amamra, Mouats, & Aouf, 2014), and (Pham, Vo, Hung, & Bac, 2010), is to process an entire frame concurrently by assigning a single thread to each pixel location. In addition to this however, other optimization methods must be used in order to maximize computational efficiency.

### 4.1 Page-Locked memory and Cuda Streams

By far the most pressing issue in using both CPU and GPU processing for a single program is memory transfer. More specifically, copying from the host memory to the device memory or vice-versa can have a latency large enough to even *slow down* the processing speed of a single program. For this reason, the proposed method makes use of pinned, or page-locked, memory. Using page-locked memory on the host side ensures that any memory being copied from the host to the device is not swapped out of host memory ("Choosing Between Pinned and Non-Pinned Memory", n.d.). The benefit of using this memory is that systems will often have more memory bandwidth in host-device transfers and it allows for memory transfers to be executed concurrently with kernel execution ("Cuda C Programming Guide", 2017). CUDA streams are objects used to manage the concurrent execution of CUDA code, which in this case is concurrent memory transfers and kernel executions. In the proposed method, streams are used such that as the kernel for frame $F_t$ is being executed, the information corresponding following frame $F_{t+1}$ is being sent to the GPU simultaneously. Around the time that the kernel for frame $F_t$ completes, the memory transfer from the device back to the host is executed concurrently with the kernel execution for frame $F_{t+1}$. The result of this is a partial masking of the latency caused by memory transfers from the host to the device (Zhang, Tabkhi, & Schirner, 2014).

## 4.2 Global Memory Accessing

Another major bottleneck in GPU execution is global memory access. Global memory can only be accessed in 32, 64, or 128 byte transactions. Warps coalesce the global memory access of their corresponding threads into one or more of these transactions, the ideal being one transaction per warp ("Cuda C Programming Guide", 2017). For this reason, the proposed method converts each frame into a four channel image, (BGRR), so that each thread requests for four bytes of information in global memory, resulting in a single 128 byte transaction for each warp and an increase in computational efficiency (Liu, 2015).

## 5 Experimental Results

For the results, we used two different video sequences from the Changedetection.net 2012 data-set, the PETS2006 video and the smaller highway video (Goyette, Jodoin, Porikli, Konrad, & Ishwar, 2012). This data-set provides a convenient benchmarking standard that is widely used and closely resembles situations encountered in real life situations, an overview can be found in (Goyette et al., 2012). We also gathered an HD video for benchmarking purposes. We tested both the original CPU implementation and the GPU implementation together and compared the results. The benchmarks were taken using a 6 core Intel i7-5820k CPU @ 3.30GHz and a GTX 980 GPU on Windows 10 64-bit. Each test was taken using a single CPU thread.

| Video | CPU Implementation | GPU Implementation | Speedup |
|---|---|---|---|
| Highway (320x240) | 882.82 FPS | 3834.93 FPS | 4.34x |
| PETS2006 (720x576) | 155.44 FPS | 849.26 FPS | 5.46x |
| HD video (1920x1080) | 21.94 FPS | 179.20 FPS | 8.17x |

### 5.1 Our Parameters

The parameters we used for our core algorithm are as follows. We set our bandwidth to 42.67 and $th = 0.03$, which proved to provide the speed and accuracy required. We updated each probability density function at every interval of 8 frames, striking a strong balance between speed and accuracy.

## 5.2 Choice of Other Methods

We used three state-of-the-art algorithms to test against our program, SOBS (Maddalena & Petrosino, 2008), ViBe (Barnich & Droogenbroeck, 2011), and the Zipfian method (Manzanera, 2007). The purpose of this comparison is to show how our algorithm performs relative to established methods, facilitating the process of judging our method's value in the field of computer vision. SOBS was chosen to serve as an example of the relative upper boundary in segmentation precision possible today, which is coupled by its often slow computational times. On the other end of the spectrum, the Zipfian method was chosen as an example of the relative upper boundary in computational time. ViBe was chosen as one of the better-established methods which seeks to strike the same balance between computational speed and precision as we do in this paper.

## 5.3 Parameters of Other Methods and Specifications

The parameters of SOBS are $e1 = 1.0$, $e2 = 0.008$, $c1 = 1.0$, $c2 = 0.05$; any parameters not listed were left at default as given in (Maddalena & Petrosino, n.d.). For ViBe, the Opencv C++ implementation was used to gather the results. We acquired the source code from (Barnich & Droogenbroeck, n.d.) and left the parameters at their default values. The parameters of the C implementation of the ZipFian method were left at the default values set by the author who kindly provided the source code. The raw TP, FP, FN, and TN numbers were computed using the C++ comparator program provided by changedetection.net. The ViBe and Zipfian methods were built from source and tested on Linux Mint 18 Cinnamon 64-Bit with Linux Kernel version 4.4.0-21-generic, while the SOBS method was tested on Windows 10 using the executable found on the author's website (Maddalena & Petrosino, n.d.).

| Category | Measurement | SOBS | Proposed Method | ViBe | ZipFian |
|---|---|---|---|---|---|
| baseline | Recall | .771 | .609 | .534 | .648 |
| baseline | Specificity | .998 | .996 | .998 | .856 |
| baseline | FPR | .001 | .003 | .001 | .143 |
| baseline | FNR | .228 | .390 | .465 | .351 |
| baseline | PWC | .902 | 1.989 | 1.954 | 15.278 |
| baseline | F-Measure | .839 | .709 | .680 | .296 |
| baseline | Precision | .920 | .868 | .994 | .232 |
| Camera Jitter | Recall | .717 | .594 | .452 | .604 |

| | | | | | |
|---|---|---|---|---|---|
| Camera Jitter | Specificity | .972 | .960 | .998 | .869 |
| Camera Jitter | FPR | .027 | .039 | .011 | .130 |
| Camera Jitter | FNR | .282 | .405 | .547 | .359 |
| Camera Jitter | PWC | 3.737 | 5.323 | 3.311 | 13.951 |
| Camera Jitter | F-Measure | .616 | .492 | .521 | .267 |
| Camera Jitter | Precision | .545 | .429 | .638 | .174 |
| Dynamic Background | Recall | .726 | .636 | .443 | .524 |
| Dynamic Background | Specificity | .985 | .979 | .997 | .966 |
| Dynamic Background | FPR | .014 | .020 | - | .033 |
| Dynamic Background | FNR | .273 | .363 | .556 | .475 |
| Dynamic Background | PWC | 1.643 | 2.448 | .822 | 3.840 |
| Dynamic Background | F-Measure | .528 | .443 | .504 | .215 |
| Dynamic Background | Precision | .467 | .402 | .671 | .153 |
| Intermittent Object Motion | Recall | .549 | .365 | .263 | .357 |
| Intermittent Object Motion | Specificity | .901 | .984 | .982 | .946 |
| Intermittent Object Motion | FPR | .098 | .015 | - | .053 |
| Intermittent Object Motion | FNR | .450 | .634 | .736 | .642 |
| Intermittent Object Motion | PWC | 11.012 | 6.136 | 6.905 | 9.638 |
| Intermittent Object Motion | F-Measure | .495 | .379 | .330 | .272 |
| Intermittent Object Motion | Precision | .554 | .691 | .674 | .269 |
| Shadow | Recall | .729 | .665 | .545 | .650 |
| Shadow | Specificity | .988 | .975 | .994 | .932 |
| Shadow | FPR | .011 | .024 | .005 | .067 |
| Shadow | FNR | .270 | .334 | .454 | .349 |
| Shadow | PWC | 2.234 | 3.737 | 2.399 | 7.912 |
| Shadow | F-Measure | .731 | .588 | .659 | .407 |
| Shadow | Precision | .740 | .581 | .874 | .305 |
| Thermal | Recall | .482 | .439 | .296 | .639 |
| Thermal | Specificity | .996 | .990 | .999 | .882 |
| Thermal | FPR | .003 | .009 | - | .117 |
| Thermal | FNR | .517 | .560 | .703 | .360 |
| Thermal | PWC | 2.641 | 5.071 | 4.591 | 13.570 |
| Thermal | F-Measure | .594 | .519 | .434 | .391 |
| Thermal | Precision | .862 | .699 | .984 | .317 |
| Bad Weather | Recall | .569 | .643 | .434 | .787 |
| Bad Weather | Specificity | .997 | .993 | .996 | .855 |
| Bad Weather | FPR | - | .006 | - | .144 |

| | | | | | |
|---|---|---|---|---|---|
| Bad Weather | FNR | .430 | .356 | .565 | .212 |
| Bad Weather | PWC | .872 | 1.277 | 1.287 | 14.630 |
| Bad Weather | F-Measure | .666 | .609 | .555 | .236 |
| Bad Weather | Precision | .835 | .618 | .857 | .166 |
| Low Framerate | Recall | .550 | .419 | .263 | .486 |
| Low Framerate | Specificity | .955 | .988 | .982 | .928 |
| Low Framerate | FPR | .044 | .011 | - | .071 |
| Low Framerate | FNR | .449 | .580 | .736 | .513 |
| Low Framerate | PWC | 5.779 | 2.873 | 6.905 | 8.238 |
| Low Framerate | F-Measure | .472 | .379 | .330 | .248 |
| Low Framerate | Precision | .548 | .451 | .674 | .184 |
| Night Videos | Recall | .603 | .491 | .283 | .638 |
| Night Videos | Specificity | .958 | .967 | .993 | .862 |
| Night Videos | FPR | .041 | .032 | .006 | .137 |
| Night Videos | FNR | .396 | .508 | .716 | .361 |
| Night Videos | PWC | 4.944 | 4.326 | 2.147 | 14.225 |
| Night Videos | F-Measure | .363 | .308 | .331 | .179 |
| Night Videos | Precision | .301 | .267 | .483 | .115 |
| PTZ | Recall | .699 | .582 | .313 | .483 |
| PTZ | Specificity | .682 | .866 | .904 | .722 |
| PTZ | FPR | .317 | .133 | .095 | .277 |
| PTZ | FNR | .300 | .417 | .686 | .516 |
| PTZ | PWC | 31.780 | 13.681 | 10.152 | 28.103 |
| PTZ | F-Measure | .040 | .195 | .089 | .033 |
| PTZ | Precision | .021 | .170 | .059 | .018 |
| Turbulence | Recall | .631 | .672 | .578 | .696 |
| Turbulence | Specificity | .994 | .971 | .999 | .898 |
| Turbulence | FPR | .005 | .028 | - | .101 |
| Turbulence | FNR | .368 | .327 | .421 | .303 |
| Turbulence | PWC | .750 | 3.013 | .269 | 10.248 |
| Turbulence | F-Measure | .463 | .234 | .674 | .062 |
| Turbulence | Precision | .438 | .163 | .822 | .034 |

## 6 Examples

Here are example frames for each method in each category, in the order of GroundTruth, SOBS, Our Method, ViBe, and ZipFian

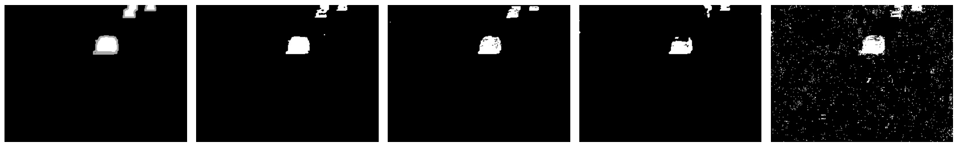

Figure 1: Baseline

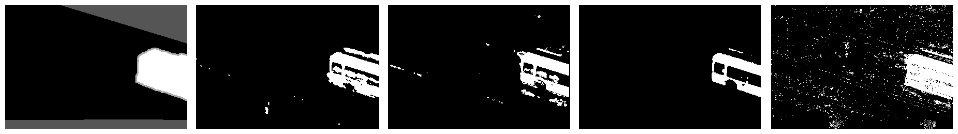

Figure 2: Camera Jitter

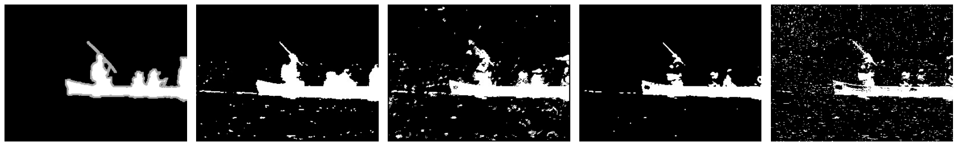

Figure 3: Dynamic Background

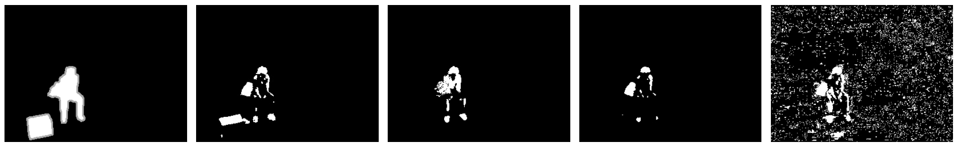

Figure 4: Intermittent Object Motion

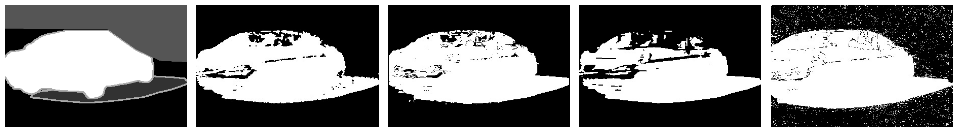

Figure 5: Shadow

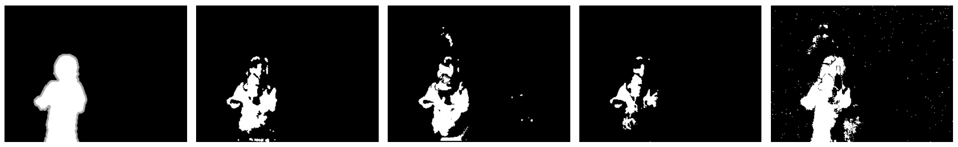

Figure 6: Thermal

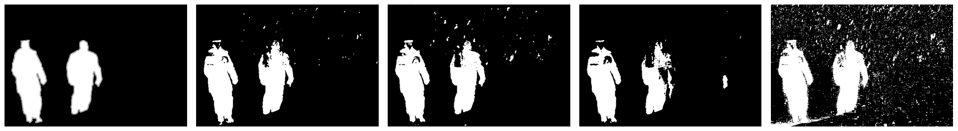

Figure 7: Bad Weather

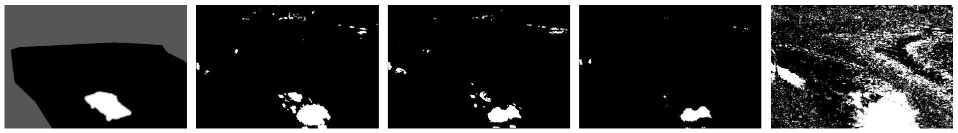

Figure 8: Night Videos

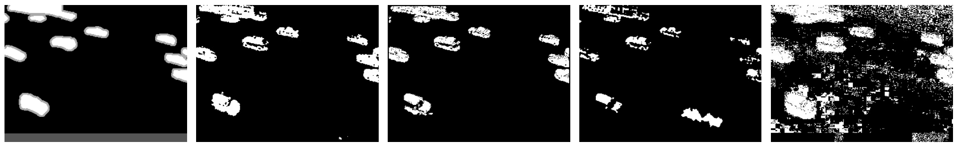

Figure 9: Low Framerate

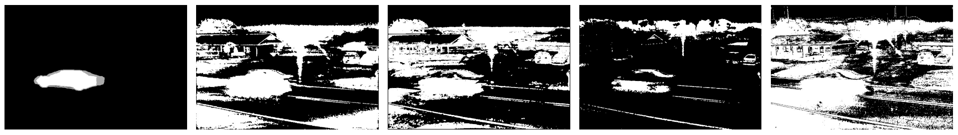

Figure 10: PTZ

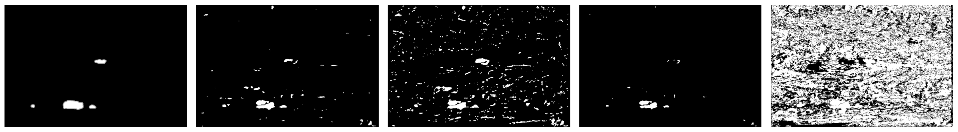

Figure 11: Turbulence

## 7 Final Thoughts

From the testing results, we can see that our GPU implementation is able to significantly improve the performance of the original CPU implementation. The speedup is not as significant at lower resolution videos, which can be attributed to the latency caused by the transfer of data from the host to the device and vice-versa. The computational power of the gpu is more readily seen in higher resolution videos, where the memory transfer latency is less prominent relative to the increased computational demand provided by the extra pixels. In addition, one could easily take the parallelization further by utilizing multiple CPU cores, all of which would run concurrent kernels relative to eachother. This method has the ability to be applied in many applications. Primarily, it shows potential for processing real time video sequences of high resolution, which would require no more than a basic system to run on.

## 8 Conclusion

Here we have proposed a GPU implementation of the background subtraction algorithm proposed in (Porr, 2017). In order to make use of the inherently parallelizable nature of background subtraction algorithms, we incorporated GPU processing using NVIDIA's CUDA computing platform. This method significantly improves upon the original cpu implementation, especially in high resolution video sequences. The proposed method also implements many memory optimization methods in order to prevent common bottlenecks caused by latency in memory transfers, including adjustments for memory coalescing and CUDA Streams. The utilization of GPU processing as done here will allow background subtraction algorithms to remain applicable as the consumer and professional markets continue to demand higher quality videos.

## References

Amamra, A., Mouats, T., & Aouf, N. (2014, Sept). Gpu based gmm segmentation of kinect data. In *Proceedings elmar-2014* (p. 1-4). doi: 10.1109/ELMAR.2014.6923325

Barnich, O., & Droogenbroeck, M. V. (n.d.). *Vibe source code, original implementation.*

Barnich, O., & Droogenbroeck, M. V. (2011, June). Vibe: A universal background subtraction algorithm for video sequences. *IEEE Transactions on Image Processing*, *20*(6), 1709-1724. doi: 10.1109/TIP.2010.2101613

Choosing between pinned and non-pinned memory [Computer software manual]. (n.d.).

Cuda c programming guide (8.0 ed.) [Computer software manual]. (2017, Jun).

Elgammal, A., Duraiswami, R., Harwood, D., & Davis, L. S. (2002, Jul). Background and foreground modeling using nonparametric kernel density estimation for visual surveillance. *Proceedings of the IEEE*, *90*(7), 1151-1163. doi: 10.1109/JPROC.2002.801448

Goyette, N., Jodoin, P. M., Porikli, F., Konrad, J., & Ishwar, P. (2012, June). Changedetection.net: A new change detection benchmark dataset. In *2012 ieee computer society conference on computer vision and pattern recognition workshops* (p. 1-8). doi: 10.1109/CVPRW.2012.6238919

Liu, D. (2015, Oct). Gpu accelerated background subtraction. In *2015 ieee 16th international conference on communication technology (icct)* (p. 372-375). doi: 10.1109/ICCT.2015.7399860

Maddalena, L., & Petrosino, A. (n.d.). *Sobs executable for windows.*

Maddalena, L., & Petrosino, A. (2008, July). A self-organizing approach to background subtraction for visual surveillance applications. *IEEE Transactions on Image Processing*, *17*(7), 1168-1177. doi: 10.1109/TIP.2008.924285

Manzanera, A. (2007). $\sigma$-$\delta$ background subtraction and the zipf law. In L. Rueda, D. Mery, & J. Kittler (Eds.), *Progress in pattern recognition, image analysis and applications: 12th iberoamericann congress on pattern recognition, ciarp 2007, valparaiso, chile, november 13-16, 2007. proceedings* (pp. 42–51). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-76725-1_5

Pham, V., Vo, P., Hung, V. T., & Bac, L. H. (2010, Nov). Gpu implementation of extended gaussian mixture model for background subtraction. In *2010 ieee rivf international conference on computing communication technologies, research, innovation, and vision for the future (rivf)* (p. 1-4). doi: 10.1109/RIVF.2010.5634007

Porr, W. (2017). *Non-parametric background detection for video surveillance.*

Zhang, C., Tabkhi, H., & Schirner, G. (2014, Sept). A gpu-based algorithm-specific optimization for high-performance background subtraction. In *2014 43rd international conference on parallel processing* (p. 182-191). doi: 10.1109/ICPP.2014.27