CUDA Implementation of an Optimal Online Gaussian-Signal-in-Gaussian-Noise Detector

Nir Nossenson and Ariel J. Jaffe

***Abstract*—We address the computationally demanding task of real time optimal detection of a Gaussian Signal in Gaussian Noise. The mathematical principles of such a detector were formulated in 1965, but a full real-time implementation of these principles was not possible for decades mainly due to technological barriers. We present a CUDA based implementation of such an optimal detector and study its decision making speed (or throughput) as function of target signal duration and signal filter length. We also compare the throughput results to those of a CPU based design. We report on detection rates ranging from 3.5 KHz for a target duration of 10756 samples up to 15.6 KHz for target duration of 92 samples. The CUDA based detector running on 384 parallel cores had a superior throughput comparing to a pure CPU implementation when target duration was longer than 600 samples.**

***Index Terms*—Gaussian Signal in Gaussian Noise, Detection, CUDA, parallel computing, Radar, Sonar, Electrophysiological Signals.**

I. INTRODUCTION

Gaussian signal in Gaussian noise is a scientific model which is used to describe a large number of real world phenomena, including radar signals [1], sonar signals [2], neural signals [3], seismological signals [4] and more (see e.g. Basseville and Nikiforov [4]). In all of the aforementioned applications, it is often required to construct an automated detector that could monitor the incoming data and sequentially decide between the presence and absence of the signal-of-interest with a minimal number of errors (i.e. an optimal detector). Often, the incoming signal is acquired at a rate of several Kilohertz or more, and the detection throughput is required to meet this speed. An example of a Gaussian signal in Gaussian Noise and the output the optimal online signal-of-interest-detector is presented in Figure 1. The implementation of the optimal detector, and its practical computational limits are at the focus of this paper.

The mathematical basis for deducing an optimal detection rule for a Gaussian signal in Gaussian noise was stated in the pioneering work of Schweppe [5], but the technology at that time could not support the realization of such a detector due to lack of computational power. As an illustrative example, Schweppe developed a simplified implementation supporting only white Gaussian samples and a target signal that lasts 4 samples. Since then, several works examined detector realizations that could support longer Gaussian target signals

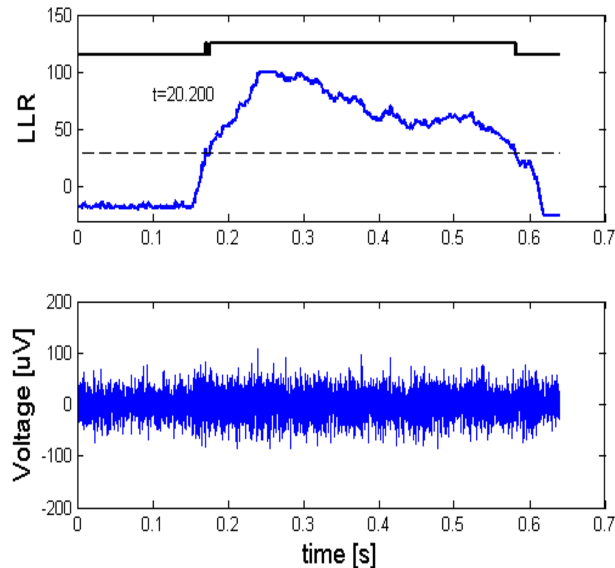


Fig. 1: An example of a Gaussian Signal in Gaussian Noise (bottom signal), together with a clear cut decision of the optimal detector (top black signal). The second signal from the top is the log likelihood ratio (LLR) produced by the optimal detector. The clear cut decision is set on when the LLR is above the dashed threshold line. To watch the multimedia demonstrating a short part of the online operation, click on the picture.

but incorporated non-optimal simplifications (see Scharf and Nolte [6], Therrien [7], Farina and Russo [8], Aloisio et al. [9], Michels et al. [10], Kulikova [11]). Specifically, these designs lacked the computationally demanding building block consisting of many chained Kalman filters which are required in the general case of long target duration with correlated signal samples. Recently, Nossenson and Messer [3] presented a full realization of a Gaussian-Signal-in-Gaussian-Noise detector which supports optimal detection of a target with *correlated consecutive* samples and target duration of *several thousands samples*. The detection performance of the latter detector was tested using artificial and real data [3, 12], and was shown to be substantially superior at very low probabilities of false alarm. However, the focus of those studies [3, 12] was detection quality, and the questions of the required computational power and decision throughput limit were not discussed.

With the appearance of the CUDA parallel computing platform [13], several authors (e.g. [14, 15]) explored the suitability of the CUDA platform for realizing detection schemes suited for other probabilistic signal models. However, for the problem of a Gaussian target signal in Gaussian noise, the

Nir Nossenson is a research associate in the Complex Dynamic Systems and Control Laboratory at Northeastern University, Boston, MA, USA. e-mail: nir.nossenson@gmail.com;n.nossenson@neu.edu.

Ariel Jaffe is with the department of mathematics and computer science at the Weizmann Institute of Science, Rehovot, Israel. e-mail:ariel.jy@gmail.com;ariel.jaffe@weizmann.ac.il.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

feasibility and throughput of a full realization of an optimal online detector has not been studied before. Thus, it is not clear whether optimal online detection of a Gaussian target signal is feasible at realistic target durations and acquisition rates.

The purpose of this paper is to asses the throughput capabilities of a fully optimal online Gaussian-Signal-in-Gaussian-Noise detector using the CUDA platform. We present a detailed CUDA implementation of the optimal Gaussian signal detector by Nossenson and Messer [3] and test its throughput limitation. We also compare our results to those achieved by a design based solely on an Intel 64 CPU.

The rest of this paper is as follows: In Section II we formulate the problem and present the basic principle of an optimal online detector. In Section III we describe the structure and realization of the detector using the CUDA technology. In Section IV, we shortly describe the method by which we assessed the throughput results. In Section V we report the speed performances of our CUDA realization as well as the speed of the reference CPU based design.

II. Preliminaries: The Gaussian Signal in Gaussian Noise Detection Problem

In the Gaussian signal in Gaussian Noise detection problem, the observed signal, $r[jT_s]$, consists of discrete incoming samples which are monitored and processed by the detector at a rate $f_s = 1/T_s$. The observed signal $r[jT_s]$, is a Gaussian random process, i.e. it can have many possible forms, but some waveform sequences are more probable than others, and the exact probability of each sequence has a Gaussian bell shape whose multi-dimensional center and radius are determined by the mean (a vector) and the covariance matrix. The acquired signal $r[jT_s]$, is further modeled as resulting from the sum of two Gaussian sources:

$$r[jT_s] = x[jT_s] + n[jT_s] \quad (1)$$

The signal $n[jT_s]$ is an additive stationary Gaussian noise with a known covariance matrix, and zero mean. The signal $x[jT_s]$ results from the target of interest. It is also a zero mean Gaussian signal, but its covariance matrix depends on the target latest appearance time, t_{target}. Had target appearance times been known, the Gaussian signal $r_0, r_1..., r_t$ would have a known covariance matrix $\Sigma_r(t_1, t_2|t_{target})$ which reflects the increased variance that the signal exhibits during target presence periods (see also Fig. 1). It is also known that after M samples from its appearance, the target is long gone and the *variance* of the observed signal $\Sigma_r(t_1, t_1)$ returns to its baseline level. The target may reappear again at an unknown time in the future, and in such case the variance is expected to rise again. Ideally, the detector should raise the detection flag for N_s samples starting from target appearance time, and should not raise the detection flag at all at other times. However, because of the inherent ambiguity of the observed signal, such an ideal detection is not possible. The optimal detector is designed such that the duration of the detection flag during target presence is as close as possible to N_s for any given average-number of false assertions of the detection flag. The basic principle of the optimal Gaussian-Signal-in-Gaussian-Noise detector is to sequentially calculate the probabilities of many hypotheses regarding the offset of the target signal given past to present observations, $\{P_1(t_{target} = now|r_0...r_t),\ P_2(t_{target} = t - T_s|r_0...r_t), ..., P_M(t_{target} = M\ samples\ ago\ or\ more|r_0...r_t)\}$. The first hypothesis corresponds to target imminent appearance ($t_{target} = t =$ now). The second hypothesis corresponds to the appearance of the target T_s seconds ago, and so forth. The last hypothesis is the Mth hypothesis which assumes that the target appeared $M \cdot T_s$ *or more* seconds ago. The detection flag is raised by the detector if the sum of probabilities corresponding to target presence is larger than the sum of probabilities corresponding to target absence, times some constant threshold value.

To calculate the probabilities $\{P_1(t_{target} = now|r_0...r_t)$,..., $P_M(t_{target} = M\ samples\ ago\ or\ more|r_0...r_t)\}$, we make use of algorithms by Kalman [16] and Schweppe [5], which facilitate sequential updating of these probabilities based on their current values and the recent incoming sample. The sequential updating procedure requires to first present the covariance matrices of the signals $x[jT_s]$ and $n[jT_s]$ in terms of auxiliary and a-priori known constant matrices and vectors of *finite* size: A, $\{\Sigma_{r,1|M}, \Sigma_{r,2|1}, ..., \Sigma_{r,M|M-1}\}$, and $\{K_1, K_2, ...K_M\}$. A brief description regarding the sizes and the meaning of these auxiliary vectors and matrices is given in Table I. Throughout the paper we assume these quantities are already known[1].

TABLE I: Summary of the problem variables and parameters.

Name	Description
$r[jT_s]$	The incoming data sample at time $t = jT_s$. The signal r streams in constantly and may include several target appearance events.
T_s	The time interval between consecutive samples. The incoming data rate is the inverse of this number: $f_s = 1/T_s$.
M	The total number of samples from target appearance until it is long gone. M is also the total number of offset hypotheses. After its disappearance, the target may re-appear at an unknown time.
N_s	The duration of an ideal detection flag assertion from target appearance, given in number of samples.
A	A constant matrix of size $2 \cdot Q \times 2 \cdot Q$ which contains the autoregressive terms of the state space representation of the processes $x[jT_s]$ and $n[jT_s]$.
$K_1...K_M$	M constant vectors (Kalman gain vectors) each of size $2 \cdot Q \times 1$.
$\Sigma_{r,1\|0}$, $\Sigma_{r,2\|1}$,... $\Sigma_{r,M\|M-1}$	M constant scalars, each representing the variance of the next sample given all past-to-present data but under a different target appearance time hypothesis. The design includes logarithmic and inverse versions of these values.
Q	The number of recursion equations required to accurately describe the target signal process and the independent noise process (the dimension of the so called "space state"). The total size of the space state matrix A is therefore $2Q \times 2Q$.

[1]An example for constructing space state matrices for Gaussian neural signals exists in Nossenson and Messer [3].

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

III. Detector Structure and its Realization

A. Operation Overview and General Architecture

As illustrated in Fig. 2, the detector implementation presented in this paper is based on a system that includes a host-CPU (central processing unit), together with a CUDA-GPU card [13] that is controlled by the CPU. The key idea of the design is the use of a GPU based parallel computation to calculate the probabilities of many hypotheses regarding target appearance time, as illustrated by text box 2 in Fig. 2. The CPU then uses these probabilities to produce a clear cut decision regarding target presence.

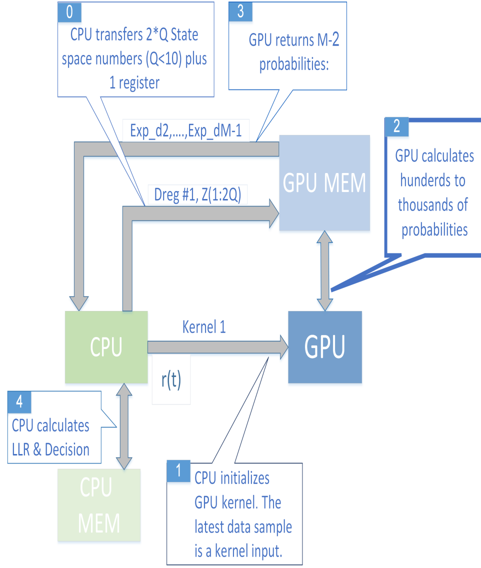


Fig. 2: Detector high level implementation

The detection process is performed online, ideally, at the rate of which the data enters the system. On the entry of every new data sample, a detection cycle begins, and the CPU sends the incoming data sample along with kernel activation commands to initiate the calculations of the hypotheses probabilities. The activation process by the CPU, and the calculation of the probabilities by the GPU are marked in Fig. 2 by text boxes 1 and 2, respectively. In the third step, the CPU reads the resulting probabilities. In the fourth and last step, the CPU uses the hypotheses probabilities to generate the log-likelihood ratio (LLR) as well as a hard decision regarding target' presence. The CPU has an additional role of calculating the probabilities of two target-offset-hypotheses that are not calculated by the GPU. The results of these calculations by the CPU are partly reflected in Fig. 2 by text box 0, and will be addressed in more depth in the next sections. Steps 0-4 are executed repeatedly, in the order implied by the text box number, as long as the data flows in.

Figure 3 depicts a more detailed block diagram of the design. The dotted green blocks are segments executed by the CPU, whereas the blue blocks are executed by the GPU using parallel computations. The core element of the design depicted in Fig. 3 is the CUDA kernel labeled as GPU kernel-1 which calculates the probabilities of many hypotheses regarding the target temporal offset given past-to-present observations. The number of parallel hypotheses (threads) is $M-2$ and it depends on the number of possible temporal offsets that the target may have. This kernel is described in more detail in Section III-B. The remaining elements are calculated by the CPU and are marked in Fig. 3 in dotted green. These computations mainly concern : a) The summation of M hypotheses probabilities into two probabilities corresponding to target presence and absence, and b) The generation of the likelihood ratio and a clear cut decision regarding target presence. These CPU operations are explained in more detail in Section III-C.

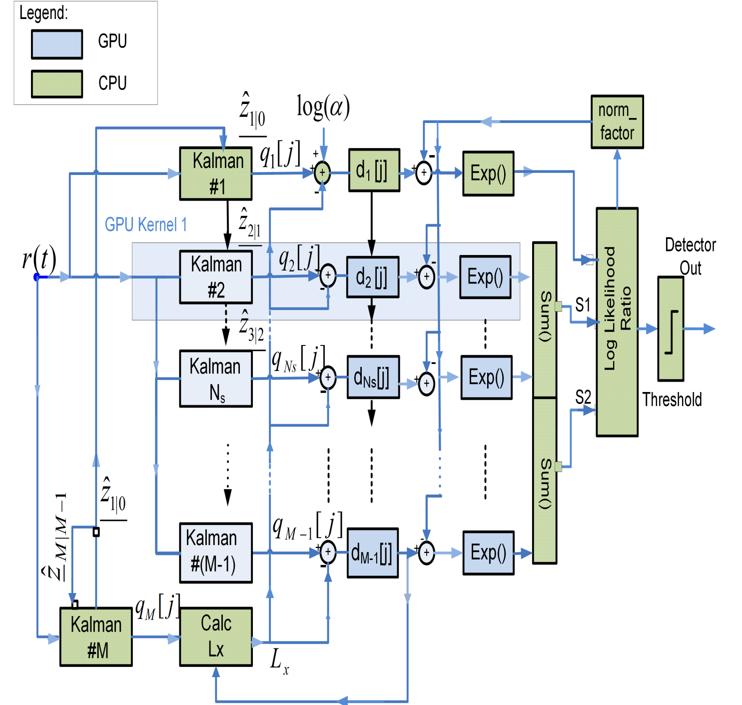


Fig. 3: Detector detailed implementation

B. GPU Kernel 1: Parallel Computation of Hypotheses Probabilities

As shown in Fig. 3, GPU Kernel 1 is the code portion which produces the probabilities of the various hypotheses regarding the temporal offset of the target signal. The same GPU kernel is executed by M-2 threads, where M is the number of different hypotheses regarding the temporal offset of the target. The threads execute the same routine but the parameters and memory used by each thread are different. Technically, the retrieval of the different memory segments is achieved by adding an address offset to the memory read by each kernel, based on the thread identity number. Given the most recent observation, $r(t)$, the kernel produces the set $\{exp_d_2, ..., exp_d_{M-1}\}$ which are normalized probabilities, each corresponding to the hypothesis that the target appeared k samples ago given past to present observations $\{r(0), ..., r(t)\}$.

To produce $\{exp_d_2, ..., exp_d_{M-1}\}$, *each thread* preforms the algorithm [3, 5] given by equations (2)-(7):

$$\hat{r}_{in,k} = \underbrace{\underline{z_{k|k-1}}_{\{1\}}}_{\substack{The\ first\\ elelemt\ of\\ the\ vector\\ z_{k|k-1}}} + \underbrace{\underline{z_{k|k-1}}_{\{Q+1\}}}_{\substack{The\ Q+1\\ element\ of\\ the\ vector\\ z_{k|k-1}}} \tag{2}$$

$$\underline{z_{k|k}} = \underline{z_{k|k-1}} + K_k \cdot (r[t] - \hat{r}_{in,k}) \tag{3}$$

$$\underline{z_{k+1|k}} = A \cdot \underline{z_{k|k}} \tag{4}$$

$$q_k[j] = \frac{1}{2}\Big[-\underbrace{\log\{2\pi\left|\Sigma_{r,k|k-1}\right|}_{see\ Table\ 1} - \left(\hat{r}_{in,k} - r_{in}\right) \cdot \underbrace{\Sigma^{-1}_{r,k|k-1}}_{see\ Table\ 1}[j\Delta_t] \cdot \left(\hat{r}_{in,k} - r_{in}\right)\Big] \tag{5}$$

$$d_k[j] = d_{k-1}[j-1] + q_k[j] - L_X[j] \tag{6}$$

$$exp_d_k[j] = exp(d_k[j] - norm_factor) \tag{7}$$

where:

- k is the offset hypothesis index ($k \in [2, .., M-1]$) which is determined by the GPU thread and block id numbers (tid and bid, respectively):

$$k = threads_per_block * bid + tid + filter_base_idx \tag{8}$$

  For example, $k = 3$ refers to the hypothesis that the target appeared 3 samples ago.
- A, K_k, $\Sigma_{r,k|k-1}$ consist of a-priori known constants. See Table I for details on their dimensions and meaning.
- $\underline{z_{k|k-1}}$ are internal state vectors that are kept in the GPU memory and are updated every sample. Their current value is used for producing $exp_d_k[j]$. There are M such internal vectors, each of size $2Q \times 1$.
- $d_k[j]$ are also internal state variables. Each of them holds a *logarithmic* version of the normalized probability to be in a certain temporal offset from target appearance time. Thus, $exp(d_k[j])$ is a normalized probability.
- $L_X[j]$ is an identical input to all the threads in the kernel. It is a normalization quantity which is produced by the CPU every clock cycle as shall be explained by eq. (9) in Section III-C.

The mathematical derivation of equations (2)-(5) and (6)-(7) was given in [5] and [3], respectively. Here we briefly emphasize three underlying conceptual ideas. The first concept is the propagation of the vectors in time $z_{k|k-1} \rightarrow z_{k+1|k}$ and $d_k \rightarrow d_{k+1}$ as described in equations (3)-(4) and (6) (see also the arrows in Fig. 3) . This propagation takes place because the hypothesis that the target appeared k samples ago at the moment j, corresponds to the hypothesis that the target appeared $k+1$ samples ago at the next sample $(j+1)$. The second concept incorporated in equations (2)-(5) is the Kalman procedure [5, 16] which allows to isolate the innovative information in the most recent incoming data $r(t)$, and to produce $q_k[j\Delta_t]$ which is the probability of the latest data $r(t)$ to support the assumed offset hypothesis. The last concept is the Bayes rule which is incorporated in eq. (6) that combines the probability of the recent data ($q_k[j\Delta_t]$) together with the information from prior samples (the d_k register) to generate the probability of a specific target offset hypothesis given all past-to-present-data.

GPU Memory Organization I: The registers $z_{k|k-1}$ and d_k ($k = 1...M-1$) are stored in the GPU device memory and are read and written by different threads of Kernel 1. To guarantee that the correct register content is read before a new value is overwritten, the design maintains in the GPU memory two copies of these array variables at all times. On the arrival of the first data sample, all the threads read from copy 'A' of the memory, and write to copy 'B'. On the arrival of the next sample, the read and write direction are reversed. This process continues as long as the data streams in.

GPU Memory Organization II: Another note-worthy point concerns the order by which the vectors K_k and $\underline{z_{k+1|k}}$ of equations (2)-(5) are stored in the memory. The algebraic vector-times-scalar multiplication in eq. (3) and the matrix-times-vector operation in eq. (4) are performed row after row (row=$1, 2, ..., 2 \cdot Q$), where ideally, all the threads should execute simultaneously the same multiplication only under different hypothesis ($k = 2, 3, ..M-1$). To minimize the stalls due to memory access, the vectors $K_2, K_3, ..., K_{M-1}$ are stored in the memory such that after the first term of the vector K_2 comes the first term of the vector K_3 (and *not* the second term of K_2). The second term of the vector K_2 is stored after the first term of the last vector, K_{M-1}. This memory organization minimizes thread stalls as it makes use of the full bandwidth of the GPU memory access bus such that several threads are serviced in a single read.

C. CPU Role: Generation of the Likelihood Ratio, Clear Cut Decision Making and Calculation of First and Last Hypotheses.

In this section we describe in more depth the roles of the CPU in performing the detection algorithm. Other than managing the GPU, the CPU main tasks are to sum the hypotheses probabilities corresponding to target presence and absence, calculate the ratio between them (the LLR) and eventually reach a clear cut decision regarding target presence by comparing the LLR to a threshold. The CPU also calculates elements associated with the first and last hypotheses since the algorithm of these two cases differ from the remaining M-2 hypotheses. We describe each of these calculations in more detail next:

1) Kalman Filter #M, and the calculation of L_X : As oppose to the $M-2$ other hypotheses, the probabilities of the first (#1) and last (#M) hypotheses are calculated by the CPU and not on the GPU. The last Kalman filter is different since its internal state is fed back, as oppose to the state propagation which is performed by the M-2 filters of GPU kernel 1 (see Fig. 3). Furthermore, the probability of the last offset hypothesis is always normalized to one, and this normalization factor L_X, is also used for normalizing all the other hypotheses-probabilities. For this reason, the CPU executes a dedicated Kalman recursion (eq. (2)-(5)) for the last Kalman filter before launching GPU-Kernel-1. This calculation produces $q_M[j]$ and also an updated Kalman state

vector for the next sample, $\underline{z}_{M|M-1}[j+1]$. Using $q_M[j]$, the CPU generates an input to GPU kernel 1 designated as L_X:

$$L_X[t] = q_M[j] + \log\{1 - \alpha + \exp(d_{M-1}[j-1]\} \quad (9)$$

where, $\alpha = 10^{-9}$ is a suitable value for detection under large a-priori uncertainty regarding target re-appearance time.

2) Kalman Filter #1 and the calculation of the first Hypothesis Register, d_1: The CPU calculates the value of the first hypothesis register in two stages. First, a dedicated CPU Kalman recursion (equations (2)-(5)) uses the recent data $r(t)$ and the state vector $\underline{z}_{1|0}$ to produce new values of q_1 and $\underline{z}_{2|1}$. Then, the value of the first hypothesis register is calculated as follows:

$$d_1[j] = \log(\alpha) + q_1[j] - L_X[j] \quad (10)$$

Comparing to the other Kalman filters, the first Kalman filter has a unique wiring of its input and output state vectors: The input state vector of this filter is $\underline{z}_{1|0}$ and it originates from the output of the last Kalman filter ($\underline{z}_{M|M-1}[j-1]$) which was calculated in the previous data sampling point. The state vector going out of the first Kalman filter ($\underline{z}_{2|1}$) is copied to the GPU memory. This copied value will be used by the first GPU thread upon the arrival of the next signal sample $r[j+1]$. This unique wiring is also illustrated in Fig. 3.

3) Probabilities Summation : Upon the completion of GPU kernel 1, the CPU sums the probabilities associated with target presence and absences into the quantities S_1 and S_2, respectively:

$$S_1 = \text{target presence sum} = \sum_{k=2}^{N_s} exp_d_k \quad (11)$$

$$S_2 = \text{target absence sum} = \sum_{k=N_s+1}^{M-1} exp_d_k \quad (12)$$

4) Generation of the normalization factor : To avoid out of range numerical errors, the exp_d_k registers produced by the GPU, are normalized by the same number, $norm_factor$ every cycle (see eq. (7) and the top right side of Fig. 3). This normalization factor is generated by the CPU as follows:

$$norm_factor[j+1] = norm_factor[j] + \log\left(max\{exp_d_1[j], S_1[j], S_2[j]\}\right) + \underbrace{-20.0}_{\text{to obtain precision for numbers lower than the maximum probability.}} \quad (13)$$

(14)

5) Calculation of the Log-Likelihood-Ratio: Following the calculation of the sums S_1 , S_2 and exp_d_1 , the CPU calculates the Log-Likelihood-Ratio which is the logarithm of the ratio between 1) S_1 augmented by the first normalized hypothesis probability, and 2) S_2 augmented by the last normalized hypothesis probability:

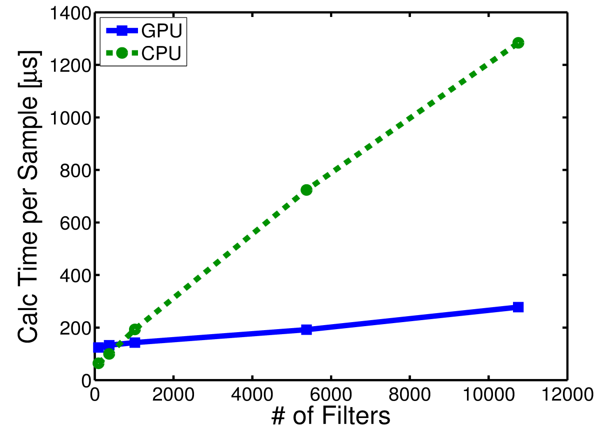
$$LLR[j] = log\big(S_1[j] + exp_d_1\big) - log\big(exp(-norm_factor[j]) + S_2[j]\big) \quad (15)$$

6) Comparison of the likelihood ratio to a threshold : A clear cut decision regarding target presence is achieved by comparing the log likelihood ratio to a threshold. The value of the threshold depends on the desired prevalence of false alarms.

IV. Measurement Methods

The CUDA detector design and a full CPU implementation which served as a reference were both tested on a Lenovo W540BG machine. Briefly, the computer included an Intel Core i7-4700MQ CPU running at speed of up to 3.40 GHz (2.4GHz minimal speed) with 4 physical cores and 8 logical cores. The GPU was NVIDIA Quadro K1100M running at 706 MHz with 2x192 cores capable of running 2x1024 threads. The computer display was handled by an additional Intel HD graphics card that served only for that purpose. The operating system was Windows-7-Ultimate. To calculate the CUDA computation time, we used the built-in measurement time utilities cudaEventRecord(start/stop) and averaged the results over 100,000 samples of input data. No other applications were ran during measurement time and the computer network connections were disabled. The pure CPU design used for comparison included the same computational building blocks of the CUDA based design (see Fig. 3), but was compiled to run solely on the CPU using MATLAB compiler. The duration of the calculation per sample was printed to a file by the compiled program and was averaged at the post-processing stage.

V. Results



M	92	364	1018	5378	10756
with GPU (μs)	124	132	143	191	278
Just CPU (μs)	64	100	193	724	1284

Fig. 4: Comparison of calculation time per incoming sample as function of the number of filters M (target duration and the number of offset hypotheses). The blue line is the GPU+CPU design, and the green dotted line is the design based solely on the CPU. A lower value corresponds to a faster calculation time. Filter length in all cases was four taps (Q=4).

Computation time per sample: Figure 4 depicts the required computation time per incoming sample as function of M

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

(the number of offset hypotheses that the target may have). The blue line reflects the computation time of the CUDA based design, whereas the green dotted line represents the design based solely on the CPU. The exact durations are also tabulated below the graph. The figure shows that for both implementations, the calculation time increases with the number of offset hypotheses in an approximately linear fashion, but with different slopes and offsets. At the lowest number of hypotheses ($M = 92$), the pure CPU is faster and completes the calculation in $64\mu s$ compared to $124\mu s$ that are required for the CUDA based design. At the largest number of offset hypotheses ($M = 10756$), the CUDA based design is more than $\times 4.5$ faster and completes the calculation in $278\mu s$ compared to $1284\mu s$ that are required for the pure CPU design. Note that for an online detection, the maximum possible incoming data rate is the inverse of the calculation time per sample. Thus, our results indicate that the maximum incoming data rates for online detection at $M = 92$ and $M = 10756$ are 15625 Hz and 3597 Hz, respectively. These results suggest that recent technological advancements facilitate optimal online detection of Gaussian signals at acquisition rates of several kilohertz and target durations lasting for several seconds. The results also suggest that the combined CPU+GPU CUDA design is much faster when the number of hypotheses are at the order of thousand hypotheses or more, but slower when the number of hypotheses is lower than about six hundred.

GPU utilization and bottlenecks: Table II informs on the execution times and efficiency of GPU-Kernel-1 as function of the number of target offset hypotheses which are listed on the first row. The second row of the table lists the kernel initiation and execution times in each case. Note that the execution time is almost the same in the first three columns. This happens because in the first three cases, all the hypotheses probabilities are calculated simultaneously by (M-2) threads (see the third row of Table II) since the number of hypotheses is much lower than the theoretical 2048 simultaneous thread limit of the hardware. The low number of threads in these three cases also leads to the low issue efficiency shown in the fourth row of Table II. In the last two columns, the number of hypotheses is higher than the number of hardware threads. In these cases, the execution time increases with the number of hypotheses since only 1536 threads are executed simultaneously. It is worth noting that the maximum number of simultaneously running threads (1536) was lower than the theoretical bound (2048) and was limited by the GPU local memory size and the fact that each thread used 35 registers. When the number of hypotheses is high (last two columns), the issue efficiency stands on $46\% - 47\%$. We found that the issue efficiency bottleneck consisted mainly of memory accesses to the z, and K arrays which took two clock cycles each. This led to a pipeline stall every other cycle. We note that this was the minimal number of stalls achieved after optimizing the memory organization as described in Section III-B.

Space state size effect: Table III shows the effect of increasing, Q, which determines the size of the matrices and vectors handled by each thread. The calculation time increases linearly in Q for both designs. This happens because the Q dimension is iterated serially is both designs, as oppose to the hypotheses dimension (M) which is computed in parallel by the GPU.

TABLE II: GPU Kernel 1 computation time, and resource utilization as function of number of hypotheses $M - 2$.

M-2	90	362	1016	5376	10754
Kernel initiation+duration (μs)	$18+25$	$18+26$	$18+28$	$18+75$	$18+148$
Simultaneously running threads	90	362	1016	1536	1536
Issue efficiency	6.6%	9.7%	24%	46%	47%

TABLE III: Calculation time per incoming sample as function of the space state dimension Q, while $M = 5378$ is fixed.

Q	4	6	10
CPU+GPU (μs)	191	225	297
Just CPU (μs)	724	798	1396

VI. Summary and Conclusions

The purpose of this study was to test whether optimal online Gaussian signal in Gaussian noise is feasible at realistic throughput of several Kilohertz, using CUDA and Intel CPU technologies. We have found that both the combined CPU+GPU based design and the pure CPU based design support such online detection rates. The stand alone CPU design was faster for short target signal durations, up to about 600 samples. The combined GPU+CPU design was up to x4.5 faster at longer signal durations. The maximum detection rates achieved were 3500 Hz - 15625 Hz at target durations of 10756 and 92 samples, respectively.

References

[1] J. Marcum, "A statistical theory of target detection by pulsed radar," *Information Theory, IRE Transactions on*, vol. 6, no. 2, pp. 59–267, 1960.

[2] G. Carter, "Time delay estimation for passive sonar signal processing," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, no. 3, pp. 463–470, 1981.

[3] N. Nossenson and H. Messer, "Optimal sequential detection of stimuli from multiunit recordings taken in densely populated brain regions," *Neural computation*, vol. 24, no. 4, pp. 895–938, 2012.

[4] M. Basseville and I. Nikiforov, *Detection of abrupt changes: theory and application*. Citeseer, 1993, vol. 10.

[5] F. Schweppe, "Evaluation of likelihood functions for Gaussian signals," *IEEE transactions on Information Theory*, vol. 11, no. 1, pp. 61–70, 1965.

[6] L. L. Scharf and L. W. Nolte, "Likelihood ratios for sequential hypothesis testing on markov sequences," *Information Theory, IEEE Transactions on*, vol. 23, no. 1, pp. 101–109, 1977.

[7] C. W. Therrien, "A sequential approach to target discrimination," *Aerospace and Electronic Systems, IEEE Transactions on*, no. 3, pp. 433–440, 1978.

[8] A. Farina and A. Russo, "Radar detection of correlated targets in clutter," *Aerospace and Electronic Systems, IEEE Transactions on*, no. 5, pp. 513–532, 1986.

[9] V. Aloisio, A. Di Vito, and G. Galati, "Optimum detection of moderately fluctuating radar targets," *IEE Proceedings-Radar, Sonar and Navigation*, vol. 141, no. 3, pp. 164–170, 1994.

[10] J. Michels, P. Varshney, and D. Weiner, "Multichannel signal detection involving temporal and cross-channel correlation," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 31, no. 3, pp. 866–880, 1995.

[11] M. V. Kulikova, "Likelihood gradient evaluation using square-root covariance filters," *Automatic Control, IEEE Transactions on*, vol. 54, no. 3, pp. 646–651, 2009.

[12] N. Nossenson, A. Magal, and H. Messer, "Detection of stimuli from multi-neuron activity: Empirical study and theoretical implications," *Neurocomputing*, vol. 174, pp. 822–837, 2016.

[13] N. Corporation. (2013) Cuda c programming guide v5. 5. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-archive

[14] N. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*. IEEE, 2009, pp. 103–109.

[15] A. Herout, R. Jošth, R. Juránek, J. Havel, M. Hradiš, and P. Zemčík, "Real-time object detection on cuda," *Journal of Real-Time Image Processing*, vol. 6, no. 3, pp. 159–170, 2011.

[16] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Fluids Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE