J Real-Time Image Proc

# Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing

**Yuliya Tarabalka · Trym Vegard Haavardsholm · Ingebjørg Kåsen · Torbjørn Skauli**



**Abstract** Hyperspectral imaging, which records a detailed spectrum of light arriving in each pixel, has many potential uses in remote sensing as well as other application areas. Practical applications will typically require real-time processing of large data volumes recorded by a hyperspectral imager. This paper investigates the use of graphics processing units (GPU) for such real-time processing. In particular, the paper studies a hyperspectral anomaly detection algorithm based on normal mixture modelling of the background spectral distribution, a computationally demanding task relevant to military target detection and numerous other applications. The algorithm parts are analysed with respect to complexity and potential for parallellization. The computationally dominating parts are implemented on an Nvidia GeForce 8800 GPU using the Compute Unified Device Architecture programming interface. GPU computing performance is compared to a multicore central processing unit implementation. Overall, the GPU implementation runs significantly faster, particularly for highly data-parallelizable and arithmetically intensive algorithm parts. For the parts related to covariance computation, the speed gain is less pronounced, probably due to a smaller ratio of arithmetic to memory access. Detection results on an actual data set demonstrate that the total speedup provided by the GPU is sufficient to enable real-time anomaly detection with normal mixture models even for an airborne hyperspectral imager with high spatial and spectral resolution.

**Keywords** Anomaly detection · Hyperspectral imagery · Multivariate normal mixture model · General purpose GPU processing

Y. Tarabalka · T. V. Haavardsholm · I. Kåsen · T. Skauli (✉)
Norwegian Defence Research Establishment (FFI),
P.O. Box 25, 2007 Kjeller, Norway
e-mail: torbjorn.skauli@ffi.no

Y. Tarabalka
e-mail: Yuliya.Tarabalka@gipsa-lab.inpg.fr

T. V. Haavardsholm
e-mail: trym.haavardsholm@ffi.no

I. Kåsen
e-mail: ingebjorg.kasen@ffi.no

## 1 Introduction

Hyperspectral imaging is characterized by its ability to record detailed information about the spectral distribution of the received light. Hyperspectral imaging sensors typically measure the energy of the received light in tens or hundreds of narrow spectral bands in each spatial position in the image, so that each pixel in a hyperspectral image can be represented as a high-dimensional vector containing the sampled spectrum. Since different substances exhibit different spectral signatures, hyperspectral imaging is a well-suited technology for numerous remote sensing applications including target detection.

When no information about the spectral signature of the desired targets is available, a popular approach for target detection is to look for objects that deviate from the typical spectral characteristics in the image. This approach is commonly referred to as anomaly detection [17], and is related to what is often called outlier detection in statistics. If targets are small compared to the image size, the spectral characteristics in the image are dominated by the background. An important step in anomaly detection is therefore often to compute a metric for correspondence with the background, which then can be thresholded to detect objects that are unlikely to be background objects.

Hyperspectral imaging inherently produces large volumes of data which create challenges in data transfer, storage and processing. In particular, real-time processing of hyperspectral imagery is no trivial task. Nevertheless, it is highly desirable in target detection and other applications to process images in real time, usually on board the platform carrying the sensor.

Several real-time anomaly detection methods suitable for on-board processing exist, like the SSRX implemented in the ARCHER and WAR HORSE programs [18, 19], but these are usually based on very simple geometric or statistical representations of the image background variability. In contrast, mixture models, such as the multivariate normal mixture model, may be able to represent the background variability quite accurately, resulting in statistically meaningful background metrics. The characteristics of anomaly detection based on normal mixture models are discussed in some detail in [5]. This anomaly detector has demonstrated good detection performance on several occasions. One of the main criticisms of this method, however, has been that it is computationally very expensive, and therefore poorly suited for on-board real-time target detection.

Fortunately, some of the most time-consuming tasks in the normal mixture model processing are easily parallelized, so that the multi-core architecture in modern central processing units (CPUs) may be exploited to speed up the processing. An interesting recent development has been the introduction of fully programmable graphics processing units (GPUs) together with software interfaces like NVIDIA CUDA [12] and AMD CTM [1] dedicated to general purpose processing on video cards. Because the GPU architectures are optimized for massively parallel processing, modern commodity video cards can achieve very high computational performance for parallel problems, peaking at several hundred GFLOPS or more. The high demand for realistic graphics (and physics) in the computer game market drives the development of increasingly powerful GPUs at low cost, while keeping computer architectures adapted to this technology to achieve very high bandwidth communication between the computer and the graphics hardware. Today, low-cost, low-weight gaming computers are readily available with extremely powerful parallel computing performance. This kind of hardware is therefore very well suited for on-board processing in a hyperspectral target detection scenario.

Although general-purpose computing on graphics processing units (GPGPU) has been an active area of research for decades, the introduction of Compute Unified Device Architecture (CUDA) and CTM has finally brought it within reach of a broader community, giving programmers access to dedicated application programming interfaces (APIs), software development kits (SDKs) and GPU-enabled C programming language variants.

This paper will consider the parallelization of an anomaly detection algorithm based on the multivariate normal mixture model and the resulting parallel GPU implementations using CUDA. These implementations will be compared to an optimized multi-core CPU implementation, and processing performance will be evaluated for different parameters. Finally, by performing a simple anomaly detection experiment in a search and rescue scenario on a real pre-recorded hyperspectral image, it is shown that parallelization of the problem and the latest developments in GPU design have made real-time on-board normal mixture based anomaly detection feasible.

The outline of the paper is as follows: in Sect. 2 the anomaly detection algorithm is presented. Section 3 discusses the parallelization of parts of this algorithm, while Sect. 4 considers the resulting parallel implementations. Experimental results are discussed in Sect. 5 and the final conclusions are presented in Sect. 6.

## 2 Anomaly detection algorithm

The anomaly detection algorithm used here is based on a global multivariate normal mixture model representation of the background clutter, as discussed in [5]. The basic steps in this processing are:

**Algorithm 1** Anomaly detection

**Input:** A hyperspectral image

1. **Estimation**: Estimate a background model by fitting a multivariate normal mixture model to a spatial subset of the image.
2. **Evaluation**: Calculate a background probability value for all pixels in the image based on the estimated model.
3. **Detection**: Detect anomalous pixels by thresholding on low background probability values.
4. **Segmentation**: Merge fragmented pixel detections to objects by performing a binary morphological closing[1] in the thresholded image.

The first two steps are the key elements in this method and also by far the most time consuming. The last two steps are considered here as post-processing, and will only be performed when evaluating detection performance. Since it is reasonable to assume that the detection and segmentation steps give insignificant contributions to the overall

[1] Morphology is discussed in most image processing textbooks, e.g. Section 8.4 in [4].

processing time, only the time spent on performing estimation and evaluation are considered in the following experiments.

Hyperspectral sensors usually record the images line by line in a "pushbroom" scanning mode. The simplest way to employ the above anomaly detection algorithm in a real-time application is to process the continuously recorded data in blocks, similar to what is done in the ARCHER system [19]. Each newly recorded block may thus be sent off to processing, provided that processing of the previous block is finished. If the processing rate is faster than the sensor acquisition rate, this results in a small latency equal to the time it takes to record a block of data. The crucial factor in enabling a real-time implementation of this algorithm is therefore to ensure that the normal mixture estimation and evaluation steps are performed faster than the time it takes to record a block of data. The following section will give a detailed explanation of the estimation and evaluation steps.

### 2.1 Normal mixture model estimation and probability value calculation

A hyperspectral image can be considered as a set of pixel vectors $\mathbf{X} = \{\mathbf{x}_j \in \mathbf{R}^B,\ j = 1,\ 2,\ ...,\ n\}$, where $n$ is the number of image pixels and $B$ is the number of spectral bands (see Fig. 1).

A multivariate normal mixture model is represented by the probability density function:

$$p(\mathbf{x}) = \sum_{c=1}^{C} \omega_c \phi_c(\mathbf{x}; \boldsymbol{\mu}_c, \Sigma_c) \tag{1}$$

where $\omega_c \in [0,\ 1]$ is the mixing proportion (or weight) of component $c$ with $\sum_{c=1}^{C} \omega_c = 1$, and $\phi(\boldsymbol{\mu}, \Sigma)$ is the multivariate normal density with mean $\boldsymbol{\mu}$ and covariance matrix $\Sigma$ :

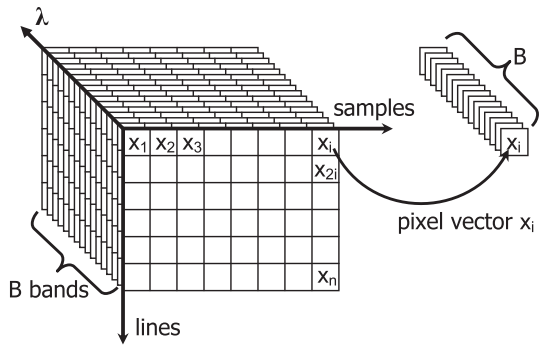


**Fig. 1** Structure of the hyperspectral image data

$$\phi_c(\mathbf{x}; \boldsymbol{\mu}_c, \Sigma_c) = \frac{1}{(2\pi)^{B/2}} \frac{1}{|\Sigma_c|^{1/2}} \times \exp\left\{ -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_c)^{\mathrm{T}} \Sigma_c^{-1} (\mathbf{x} - \boldsymbol{\mu}_c) \right\}. \tag{2}$$

Estimating a multivariate normal mixture model for the background is therefore equivalent the problem of estimating the parameters $\boldsymbol{\psi} = \{C, \omega_c, \boldsymbol{\mu}_c, \Sigma_c;\quad c = 1, 2, \ldots, C\}$, given a set of image data. The total number of parameters that must be estimated is $P = (B(B+1)/2 + B + 1)C + 1$ which in typical hyperspectral anomaly detection applications may be a quite large number. But since the background model estimation is based on data in the entire image block under consideration, more than enough data are available for the estimation process. In fact, the amount of data available may exceed that needed to make a statistically significant estimation of the model parameters. To avoid wasting time on processing more data than necessary, a subset of pixel vectors $\mathbf{S} = \{\mathbf{s}_j \in \mathbf{R}^B, j = 1, 2, \ldots, m\}$, $\mathbf{S} \subseteq \mathbf{X}$, is considered where $m$ is the number of pixels in the subset.

The actual estimation procedure used in this paper is an iterative method similar to the SEM algorithm [8], as outlined in Algorithm 2. The principal idea is to assume that each pixel $\mathbf{s}_j$ from subset $\mathbf{S}$ belongs to one of the components $c = 1, 2, \ldots, C$. Thus, on each iteration $i$ we obtain a partition $\mathbf{Q}_1^i, \mathbf{Q}_2^i, \ldots, \mathbf{Q}_C^i$ of the subset $\mathbf{S}$, where $\mathbf{Q}_c^i = \{\mathbf{x}_{j,\ c}^i \in \mathbf{R}^B,\ j = 1, 2, \ldots, m_c^i\}$ contains the pixels belonging to the component $c$ on the iteration $i$, and $m_c^i$ is the number of pixels in $\mathbf{Q}_c^i$.

**Algorithm 2** Estimation step

**Input:**

- a subset $\mathbf{S}$ of hyperspectral pixels
- an upper bound $\mathrm{X}_{max}$ on the number of components
- a threshold $\delta$ for termination of the iteration process
- a maximum number of iterations $\mathrm{I}_{max}$

**Output:** The model parameters $\boldsymbol{\psi}$.

*Initialization (Iteration 0):*

Let $C = C_{max}$. Determine the first partition $\mathbf{Q}_c^0, c = 1, 2, ..., C$ of $\mathbf{S}$:

1. Choose randomly C pixels from the subset $\mathbf{S}$ to serve as component (cluster) centers (Fig. 2, Task 1).
2. Assign pixels of the subset $\mathbf{S}$ to the components on the basis of the nearest Euclidean distance to the component center (Fig. 2, Task 2).

**For every iteration $i > 0$ ($I$ iterations in total):**

*Parameter estimation step:*

Estimate $\boldsymbol{\mu}_c^i$, $\boldsymbol{\Sigma}_c^i$ and $\omega_c^i$ for $c = 1, 2, ..., C$, using the component-wise Maximum Likelihood estimates (Fig. 2, Tasks 3-5):

$$\boldsymbol{\mu}_c^i = \frac{1}{m_c^{i-1}} \sum_{j=1}^{m_c^{i-1}} \mathbf{x}_{j,c}^{i-1} \tag{3}$$

$$\boldsymbol{\Sigma}_c^i = \frac{1}{m_c^{i-1}} \sum_{j=1}^{m_c^{i-1}} (\mathbf{x}_{j,c}^{i-1} - \boldsymbol{\mu}_c^i)(\mathbf{x}_{j,c}^{i-1} - \boldsymbol{\mu}_c^i)^T \tag{4}$$

$$\omega_c^i = \frac{m_c^{i-1}}{m}. \tag{5}$$

*Component assignment step:*

1. Assign each pixel in the subset $\mathbf{S}$ to one of the components (Fig. 2, Task 6) according to the maximum *a posteriori* probability criteria[2]:

$$\mathbf{x}_j \in \mathbf{Q}_c^i : \Pr(c|\mathbf{x}_j) = \max_l \Pr(l|\mathbf{x}_j) \tag{6}$$

where

$$\Pr(c|\mathbf{x}_j) = \frac{\omega_c^i \phi_c(\mathbf{x}_j; \boldsymbol{\mu}_c^i, \boldsymbol{\Sigma}_c^i)}{\sum_{c=1}^{C} \omega_c^i \phi_c(\mathbf{x}_j; \boldsymbol{\mu}_c^i, \boldsymbol{\Sigma}_c^i)}. \tag{7}$$

2. To avoid problems related to degenerations in the model, eliminate component $c$ if $m_c^i < B, c = 1, 2, ..., C$ (Fig. 2, Task 7). The pixels that belonged to the deleted components will be reassigned to the other components in the next iteration.

3. If the number of pixels from the subset $\mathbf{S}$ that changed component membership is larger than the threshold $\delta$ and the number of iterations has not exceeded the maximum number of iterations $I_{max}$, return to the parameter estimation step.

Having estimated the multivariate normal mixture model for the background, a metric for correspondence with the background is calculated for each pixel in the hyperspectral image $\mathbf{X}$ by evaluating the model probability density value for each pixel spectrum, as outlined in Algorithm 3 (see also Fig. 2, Task 8).

**Algorithm 3** Evaluation Step

**Input:**

- a hyperspectral image $\mathbf{X}$
- a multivariate normal mixture model for the background $p(\mathbf{x}; \boldsymbol{\psi})$

**Output:** $p_{bg}(j)$

For each hyperspectral pixel $\mathbf{x}_j$ in $\mathbf{X}$, calculate the background probability value

$$p_{bg}(j) = p(\mathbf{x}_j; \boldsymbol{\psi}) \tag{8}$$

[2] The original SEM algorithm uses the stochastic component assignment instead, a slower but more robust approach.

## 3 Parallelizing the anomaly detection algorithm

A block diagram of the anomaly detection algorithm is shown in Fig. 2. One of the characteristics of the algorithm is its regular (pipeline) structure. The figure gives the computational complexity for each algorithm task. We assume that the number of pixels in the original image block, as well as the subset used for the model estimation, is significantly larger than the number of components, number of bands and number of iterations in the estimation step ($n, m \gg C, B, I$). Then the overall computational complexity for the estimation step (Tasks 1–7) is $O(mCB^2I)$, and for the Evaluation step (Task 8) it is $O(nCB^2)$. Since in our case one block of hyperspectral data has spatial dimensions of order $10^3 \times 10^3$, the total number of pixels $n \sim 10^6$. We assume that $m \sim 10^5$ and $C$, $B \sim 10^1$. Generally, the computational cost of the anomaly detection algorithm is high. Thus, running the algorithm in real-time requires an efficient implementation and high-performance hardware.

Plaza et al. [13] and Setoain et al. [15] have reviewed parallel processing of hyperspectral images. There are two main approaches to decompose the problem into parts that can be run concurrently: task-level decomposition and data-level decomposition [9, 11]. Setoain et al. [15] distinguish task-level, spatial-level and spectral-level parallelism for the hyperspectral image processing algorithms (the last 2 levels are the particular cases of the data decomposition patterns).

Task-level parallelism refers to different and independent sets of instructions executing in parallel. Spatial-level parallelism decomposes the image into subsets of pixel vectors that are operated on independently, thus forming data streams processed concurrently by the processing elements (the finest level being pixel-level decomposition, when each processing element is working on 1 pixel vector). Spectral-level parallelism refers to decomposition of the multi-band image data into units containing subsets of contiguous spectral bands.

Task-level parallelization is not possible here, as Fig. 2 shows that execution of each task requires the results from the previous task. Analysing the computational complexities of the parts of the algorithm, we can distinguish those with the highest computational cost as tasks 2, 4, 6, 8, marked by ellipses in Fig. 2. Fortunately, all these four tasks can be parallelized, using data-level decomposition.

The tasks that assign pixels to the components (Tasks 2, 6), and the evaluation task (Task 8) exhibit inherent parallelism at pixel level, the finest level of spatial parallelism. This results in simple, robust, scalable and easily understandable parallel implementation of these tasks. The number of threads that can be run concurrently is equal to the number of pixels ($n$, $m$). As the values of $n$, $m$ are high, the amount of concurrency is significant. We note that all

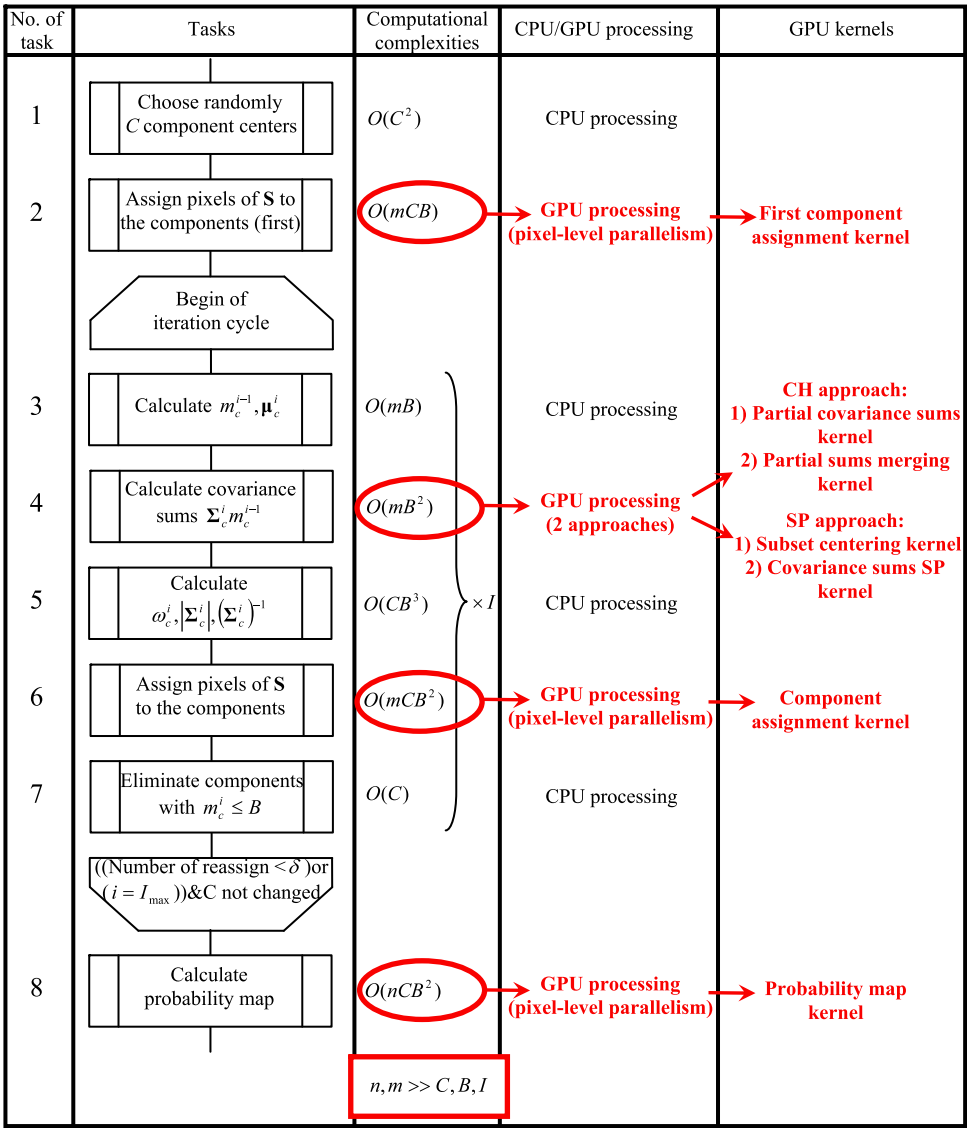


**Fig. 2** Block diagram of the anomaly detection algorithm ($n$ number of image pixels, $m$ number of pixels in the estimation subset $\mathbf{S}$, $B$ number of bands, $C$ number of components in the mixture, $I$ number of iterations). *Red ellipses* indicate tasks with the highest computational cost. The diagram also summarizes the structure of GPU-based algorithm implementations, as discussed in the text

the concurrent threads of these tasks will require the common parameter data (like weights, means, inverse covariance matrices etc.). These data remain constant and can be efficiently shared between threads.

A more challenging step is the calculation of what we call the covariance sums in task 4: $\mathbf{Z}_c^i = m_c^{i-1}\Sigma_c^i$. Here $CB(B+1)/2$ elements must be estimated (symmetric covariance sum for each of the components). Several approaches to parallelize this task are possible. We consider two approaches.

### 3.1 Covariance sums: chunking approach (CH)

The first approach splits the hyperspectral image subset $\mathbf{S}$ into $K$ parts (chunks), and calculates the covariance sums for all the parts in parallel. Subsequently, covariance sums for the whole subset are calculated by summing in parallel the covariance sums for its parts (see Algorithm 4). Figure 3b represents schematically these two branching steps.

**Algorithm 4** Covariance sums - chunking approach

1. Decompose the subset $S$ into $K$ parts $U_k, k = 1, 2, ..., \mathrm{K}$ and execute in parallel $T = K$ threads:
   **for** each thread $k$:
      **for** each pixel $\mathbf{x}_{j,c}^{i-1} \in U_k$:
         $\mathbf{z}_{j,c}^{i-1} = \mathbf{x}_{j,c}^{i-1} - \boldsymbol{\mu}_c^i$
         $\mathbf{Z}_c^i = \mathbf{Z}_c^i + (\mathbf{z}_{j,c}^{i-1})(\mathbf{z}_{j,c}^{i-1})^T$
      **end for**
   **end for**

2. Each element of the covariance sums for the subset $S$ is calculated in parallel as the sum of the corresponding elements for the $K$ chunks of the subset (in total $CB(B+1)/2$ elements, so $T = CB(B+1)/2$ threads are executed in parallel).

Regarding the scalability of the chunking approach, with the increase of the number of chunks $K$, more memory is needed to store intermediate covariance sums. Thus, there is an upper bound on $K$, and the scalability of the first step of the considered approach depends on the memory available and the memory bandwidth. The scalability of the second step is limited by the $CB(B+1)/2$ concurrent threads. However, as the first step includes multiplication operations and in total more arithmetic operations per thread than the second step (for the typical configuration of values $n$, $K$ and $B$), the complexity of the chunking approach is dominated by the first step.

### 3.2 Covariance sums: spectral-level parallelism (SP)

Another way to parallelize the covariance sums estimation is to calculate in parallel the covariance between bands $q$ and $r$ ($\mathbf{Z}(q, r)$, $q = 1, 2, ..., B$; $r = 1, ..., q$). Each thread will calculate $C$ elements $\mathbf{Z}_c^i(q, r)$, $c = 1, 2, ..., C$ (see Fig. 3c).

The algorithm consists of two branching steps: centering of the input subset $S$ (in $m$ parallel threads) and covariance sums calculation (see Algorithm 5).

**Algorithm 5** Covariance sums - spectral-level parallelism

1. **for** each pixel $\mathbf{x}_{j,c}^{i-1} \in S$ ($T = m$ concurrent threads):
   $\mathbf{z}_{j,c}^{i-1} = \mathbf{x}_{j,c}^{i-1} - \boldsymbol{\mu}_c^i$
   **end for**

2. Each concurrent thread (in total $T = B(B+1)/2$ threads) calculates $\mathbf{Z}_c^i(q, r), c = 1, 2, ..., C$.

The complexity of this algorithm is dominated by the second step, where $T = B(B+1)/2$ threads are executed concurrently. As $B \sim 10^1$, the scalability here is seriously limited. This approach is interesting when the number of bands is significant.

In Sect. 5.3 below, we compare the execution speed and scalability of the two approaches for computation of covariance sums.

## 4 GPU-based parallel implementation

The previous section has shown that several tasks of the anomaly detection algorithm possess a significant amount of data-level concurrency, suitable for a "single instruction multiple data" architecture that allows massively parallel processing.

We have chosen to implement the parallel anomaly detection algorithms on an NVidia GeForce 8800 Ultra GPU, exploiting the new CUDA technology [12]. Through

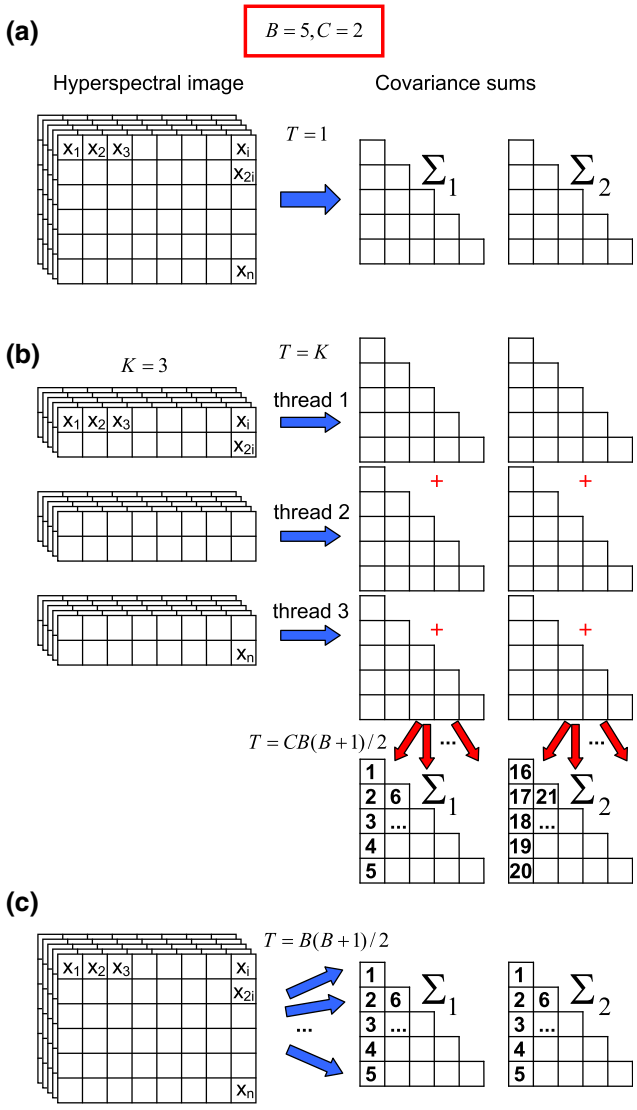


**Fig. 3** Different approaches for calculation of the covariance sums: **a** sequential algorithm; **b** parallel algorithm—chunking approach; **c** parallel algorithm—spectral-level parallelism. The figure assumes $B = 5$, $C = 2$ and $K = 3$. The *numbers* in the covariance sums' matrix cells correspond to different parallel threads

CUDA, the GPU (*device*) operates as a highly multi-threaded coprocessor to the main CPU (*host*). This means that the part of the program executed many times independently on different data can be isolated into a function (*kernel*), compiled to the device instruction set and executed concurrently on the device. The GPU is capable of running a very high number of threads in parallel.

The host and the device have their own DRAM (*host memory* and *device memory*, respectively). The data can be copied from one memory to another, by using the device's high-performance Direct Memory Access engines. This improves significantly the data transmission performance, when compared to the previous GPU programming models.

Two main conditions must be fulfilled to achieve a good performance gain:

- Overlapping of memory accesses with arithmetic operations. GPU-based implementation is well suited for problems with high arithmetic intensities (ratio of arithmetic operations to memory accesses).
- Optimization of memory accesses. The device has on-chip shared memory (that threads can use for data sharing) with very fast read and write access and off-chip constant and texture cached memories. The high-bandwidth memory use must be maximized (like shared memory, cached accesses), while minimizing the accesses to uncached memory.

From the analysis in the previous section, the anomaly detection algorithm appears to fulfill these requirements reasonably well.

The most computationally demanding tasks of the algorithm have been implemented into seven GPU kernels as summarized in Fig. 2. A brief overview of the GPU/CUDA implementation for Tasks 2, 4, 6, 8 is given below:

- *Task 2—First component assignment kernel*: Each thread determines the normal mixture component with the minimal Euclidian distance between its center and the current pixel (each thread operates on one pixel), and stores the index of this component to the component membership array. Before executing the kernel, vectors of the C component centers are copied to the device constant memory. These values are cached once and afterwards they are used by each thread from the constant cache, thus optimizing the memory access time. In total $T = m$ threads are executed in this task.
- *Task 4*:

  CH approach (refer Algorithm 4):

  1. *Partial covariance sums kernel*: Each thread calculates the covariance sums (for C components) for one (current) chunk of the subset $\mathbf{S}$ (in total $T = K$ threads), taking as input the component membership array and the means for normal mixture components. Before the kernel execution, the component means are mapped into the device texture memory (as a 2-dimensional CUDA array). These values are cached during the kernel execution. For each pixel, first a thread calculates its centered vector and store this vector to the shared memory. Then, this vector is used to calculate and add the contribution of the pixel to the covariance sum of the component, to which this pixel belongs. Each element of the vector will be read from the memory $B + 1$ times; therefore, the use of the shared memory optimizes the memory access time.
  2. *Partial sums merging kernel*: This kernel calculates covariance sums for $C$ components, by summing the $K$ partial covariance sums vectors, produced by the previous kernel. Each thread calculates one element of the covariance sums vector (which contains $CB(B + 1)/2$ elements). Thus, in total $T = CB(B + 1)/2$ threads are executed.

  SP approach (refer Algorithm 5):

  1. *Subset centering kernel*: Each thread calculates the centered vector for one pixel (in total $T = m$ threads), taking as input the means for normal mixture components (mapped into the device texture memory) and the component membership array.
  2. *Covariance sums SP kernel*: Each thread calculates $C$ elements $\mathbf{Z}_c^i(q, r)$, $c = 1, 2, ..., C$ of the covariance sums vector (see Sect. 3 for details). The kernel takes as inputs the array of centered pixel vectors, produced by the previous kernel, and the component membership array. The elements $\mathbf{Z}^i(q, r)$ are kept in the shared memory during their calculation. In total $T = B(B + 1)/2$ threads are executed.
- *Task 6—Component assignment kernel*: Each thread operates on one pixel of the subset $\mathbf{S}$ (in total $T = m$ threads), and assigns component membership according to (6). The kernel requires as inputs the parameters of the normal mixture model (weights, means and covariance matrices for $C$ components). These parameters are stored in the device texture memory. The kernel's output is the component membership array. The intermediary vectors of centered pixel values (each vector is local for each thread) are kept in the local off-chip memory. They could be put in the shared memory as well, but as the size of the shared memory is limited (16 KB per multiprocessor for an NVidia GeForce 8800 Ultra), this will limit the number of threads running concurrently. Keeping these vectors in the local memory allows to run many threads in parallel, and the memory latencies (due to the access to the off-chip memory) are hidden by multithreading.
- *Task 8—Probability map kernel*: Each thread calculates for one pixel of the hyperspectral image $\mathbf{X}$ a background probability value (1), in total $T = n$ threads. The parameters of the normal mixture model (weights, means and covariance matrices for $C$ components) stored in the texture memory are used as inputs. The vectors of centered pixel values are kept in the local

off-chip memory (the same reasoning as for the *Component assignment kernel*). The resulting probability map is an important intermediate result of the anomaly detection algorithm.

The memory usage has been carefully optimized for all kernels, so that the fast shared memory and cached memories are used intensively. However, the device memory filling will depend on the size of the hyperspectral image **X**, and the chosen subset size.

It can be noted that while CPU parallel code can be more easily adapted to different ranges of user parameters and data characteristics, the GPU code must ideally be designed for a specific problem size to have optimal performance. In our experiments, we use the same program for different ranges of parameters. Our code allows a range of reasonable parameters in the anomaly detection problem, but the performance may be sub-optimal for particular configurations of parameters.

## 5 Experimental results

### 5.1 Executing platforms and implementations

Our experiments were performed on a 2006-model HP xw8400 Workstation based on dual Quad-Core Intel Xeon processor E5345 running at 2.33 GHz with 1.333 MHz bus speed and 3 GB RAM. The computer was equipped with a XFX GeForce 8800 Ultra video card with 128 stream processors, 768 MB memory, 612 MHz core clock, 1,511 MHz shader clock and 2.16 GHz memory clock. This video card served as the primary display as well as a CUDA device.

Three different implementations of the anomaly detection algorithm have been made, one for the multi-core CPU and one GPU-based implementation for each of the covariance sum approaches (GPU-CH and GPU-SP). The CPU implementation is our performance reference, and also serves to check the precision and correctness of the GPU-based implementations.

Programs are built and run under the Windows XP 32-bit operating system. The CPU implementation is built with the Intel C++ Compiler 9.1 using OpenMP [3], BLAS [7] and LAPACK [2] libraries, while the GPU implementations has been made using the CUDA compiler driver **nvcc** [12] (CUDA Toolkit 1.0 and CUDA SDK 1.0 are used). For all implementations, the code has been carefully optimized including the mathematical representations, memory use and threading.

The dual Quad-Core Intel Xeon processor has eight cores, and therefore, up to eight threads can be executed in parallel on CPU. The parallel implementation on CPU is efficient when a few concurrent threads execute relatively large number of operations (whereas GPU parallel implementations are efficient for executing a very high number of threads concurrently).

In our reference CPU-based implementation, Tasks 6 and 8 are implemented in parallel by means of OpenMP, so that each thread operates on one pixel (the same spatial-level parallelism as for the GPU-based implementations). As the anomaly detection algorithm includes a lot of operations on vectors, BLAS functions are used intensively throughout the program to optimize the processing time. Furthermore, the determinants and inverses of covariance matrices were computed using LAPACK functions. We also tried to run in parallel other parts of the program, but for the typical range of parameters in the anomaly detection problem the processing time was not reduced.

It can be also noted that the scalability of the CPU-based implementation is seriously limited by the number of processing cores available for the program execution. Currently, the number of CPU cores cannot be increased much beyond our eight-core desktop system before weight and power consumption becomes unacceptable for on-platform processing in many important cases such as airborne applications. Furthermore, the increase of performance through the generations of recent GPUs is faster than for CPUs.

### 5.2 Hyperspectral image data set

The hyperspectral data used here originate from a real airborne hyperspectral recording of a forest scene east of Oslo, Norway. The image was captured by a HySpex [10] visual and near infrared (VNIR) hyperspectral camera from an altitude of about 1,500 m above ground level. The HySpex VNIR module is a push-broom imager covering the spectral range from 0.4 to 1.0 μm in 160 spectral bands with 1,600 spatial pixels over a 17° cross-track field of view. The acquisition rate of the camera is about 100 lines/s or 0.16 Mpixels/s.

The 1,600 by 1,200 pixel (1.92 Mpixel) block used in the following experiments is extracted from the original hyperspectral image and is spectrally downsampled to 2–50 bands by averaging over neighbouring bands. In correspondence with several investigations into the number of bands required to obtain good target detection performance [6, 16], we expect to achieve good detection results in the lower half of this interval.

The targets used in the experiments are objects considered relevant in a search and rescue scenario. They are comprised of a green canvas textile similar to that one would find in some tents, and four sets of different coloured clothing laid on the ground in the direction of the four cardinal points north, east, south and west. The targets were

placed in plain view on a small marsh. Figure 4 shows photographs of the targets and the surrounding environment, while Table 1 describes each target in more detail.

### 5.3 Basic performance assessment

We evaluate the performance of the CPU and GPU-based implementations by measuring the execution time as a function of several parameters: the number of bands $B$, the number of components in the mixture $C$, the number of pixels $m$ in the training set $\mathbf{S}$ and number of iterations $I$. Thus for our basic performance testing, the number of iterations is an input parameter and not controlled by a convergence criterion.

In the experiments we vary one parameter at a time, keeping the others fixed at the following standard configuration: $B = 15$ bands, $C = 10$ components, $I = 10$ iterations and a subset size of $m = 192{,}000$ pixels (10% of the whole image block). The execution time is measured for the complete execution as well as for individual parts. Here we report separately the contributions of the initialization part (Tasks 1–2) and the covariance matrix calculation part (Task 4) of the estimation step, and the time spent on the evaluation step (Task 8).

To determine the program execution time, the C function *clock()* was used for the CPU implementation and the CUDA timer was used to measure time for the GPU implementations. The total time measurement is started right after the hyperspectral image file is read to the CPU memory and stopped right after the resulting probability map is obtained and stored in the CPU memory. For timing of the individual parts, memory transfers related to these parts are included.

The measurements were found to be repeatable within about 1% for the GPU implementations. For the CPU implementation the variation was somewhat larger, probably due to interrupts and task scheduling by the operating system, although there is still good consistency across the explored range of parameters. For real-time applications it is interesting to note that the GPU execution time measurements are very stable. This means that the GPU may be run closer to its peak performance, with less needs for time margins compared to the CPU.

Figure 5 shows the measured total execution time when varying the different parameters. Not surprisingly, the execution time scales approximately linearly with the number of components $C$, iterations $I$, and subset size $m$. With increasing band count $B$, the increase in execution time is somewhat faster than linear. The overall result is that the GPU increases computing speed by a significant factor. The gain is particularly large for lower band counts, for example more than 20 times faster for 5 bands. At 15 bands the speedup is a factor 10, while at 50 bands a more modest factor of 3 is obtained.

The lower gain at high band count is essentially due to the covariance sums computation which becomes more memory intensive and hence less adapted to GPU processing for increasing covariance matrix dimensionality. As Fig. 6 shows, the CPU implementation performs comparable to or better than the GPU-CH implementations for most band counts during the covariance sum processing, while the GPU-SP is much slower than the other implementations below 25 bands.

Analysing the algorithms of covariance sums computation, several reasons can be suggested why the GPU-CH implementation for this task is slower than the CPU-based one. For a small number of bands the calculation time is spent mostly to run through all the array of pixel vectors.

**Table 1** Target descriptions

| Name | Description |
|---|---|
| A | Green canvas, about 1.5 × 2.5 m |
| B | Jeans jacket and pants |
| C | Grey coat |
| D | Red jacket and pants |
| E | Green jacket and pants |

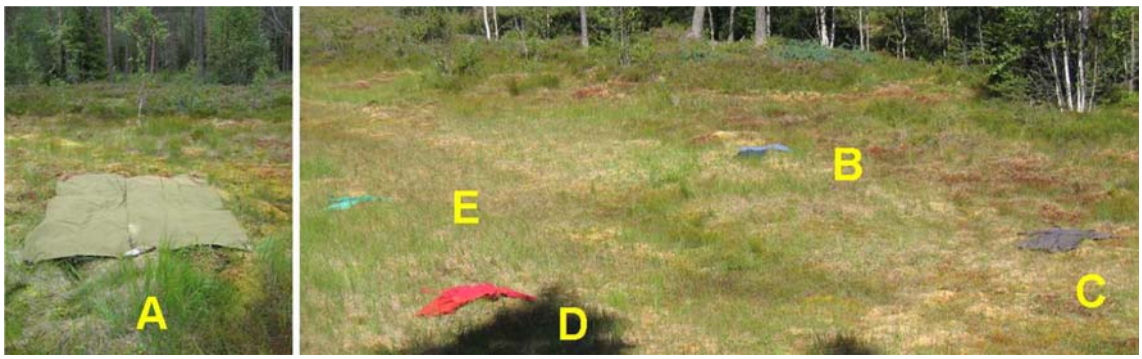


**Fig. 4** Target layout and the names used to refer to them in the following experiments. See Table 1 for more information

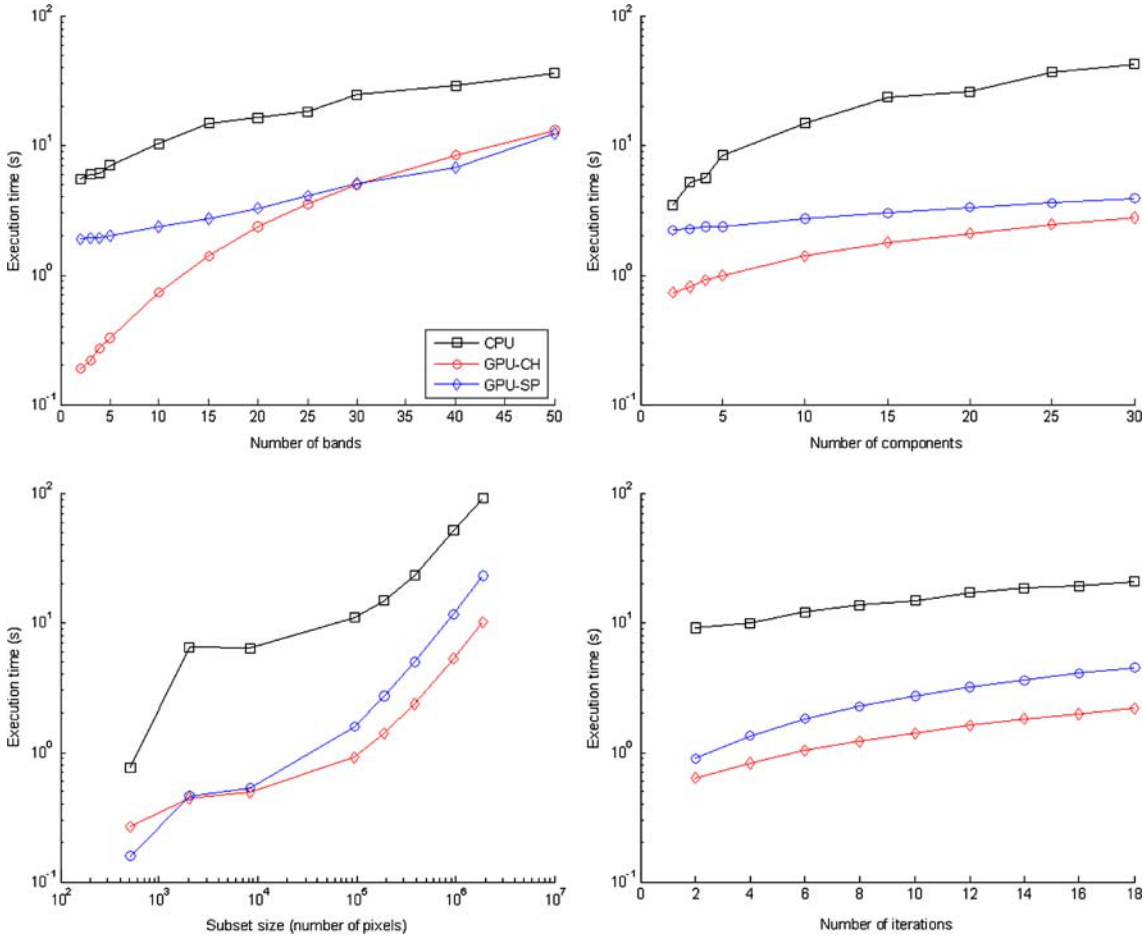


**Fig. 5** Total execution time for the three implementations when varying different parameters. The plots show the execution time for different choices of number of bands (*top left*), number of components (*top right*), subset size (*bottom left*) and number of iterations (*bottom right*). Here, the default configuration used is 15 bands, 10 components, 10 iterations and a subset size of 192,000 pixels

When we split this array of pixel vectors into several parts (chunks) in the CH approach, the GPU execution time for this parallel approach becomes faster. But when the number of bands increases, the running through bands becomes more computationally demanding. In this case:

1. More memory is needed to store covariance sums for $K$ chunks. As they are stored in the device global memory, memory bandwidth causes the increase of the processing time, when compared to CPU implementation. The processing on CPU allows data caching, which becomes especially advantageous when the number of bands increases.
2. As was mentioned before, the GPU code must be designed for a specific problem size and thread configuration to have optimal performance. A GPU kernel is executed in parallel by the batch of threads, organized as a grid of thread blocks [12]. The number of blocks and threads per block must be chosen to maximize performance. Furthermore, for the CH approach of covariance sums computation the number of chunks $K$ must be chosen. The GPU code was optimized for the standard configuration of parameters ($B = 15$ bands, $C = 10$ components, $I = 10$ iterations and $m = 192{,}000$ pixels). In particular, the number of chunks $K = 512$ was chosen by the experimental tuning and fixed in the program. As can be seen from Fig. 6, the GPU-CH implementation is the fastest for this configuration of parameters (when $B = 15$ bands, the processing time for the GPU-CH implementation is 570 versus 720 ms for the CPU implementation). If the GPU-code is adapted for another configuration of parameters, the processing speed may be increased for this particular configuration.
3. It must be noted that we varied the number of bands $B$, while keeping the estimation subset size $m$ constant. However, with increasing $B$, the number of parameters of the multivariate normal mixture model increases, and larger subset of pixels is needed to obtain an accurate estimate of parameters. When varying the subset size $m$ together with the number of bands $B$, the

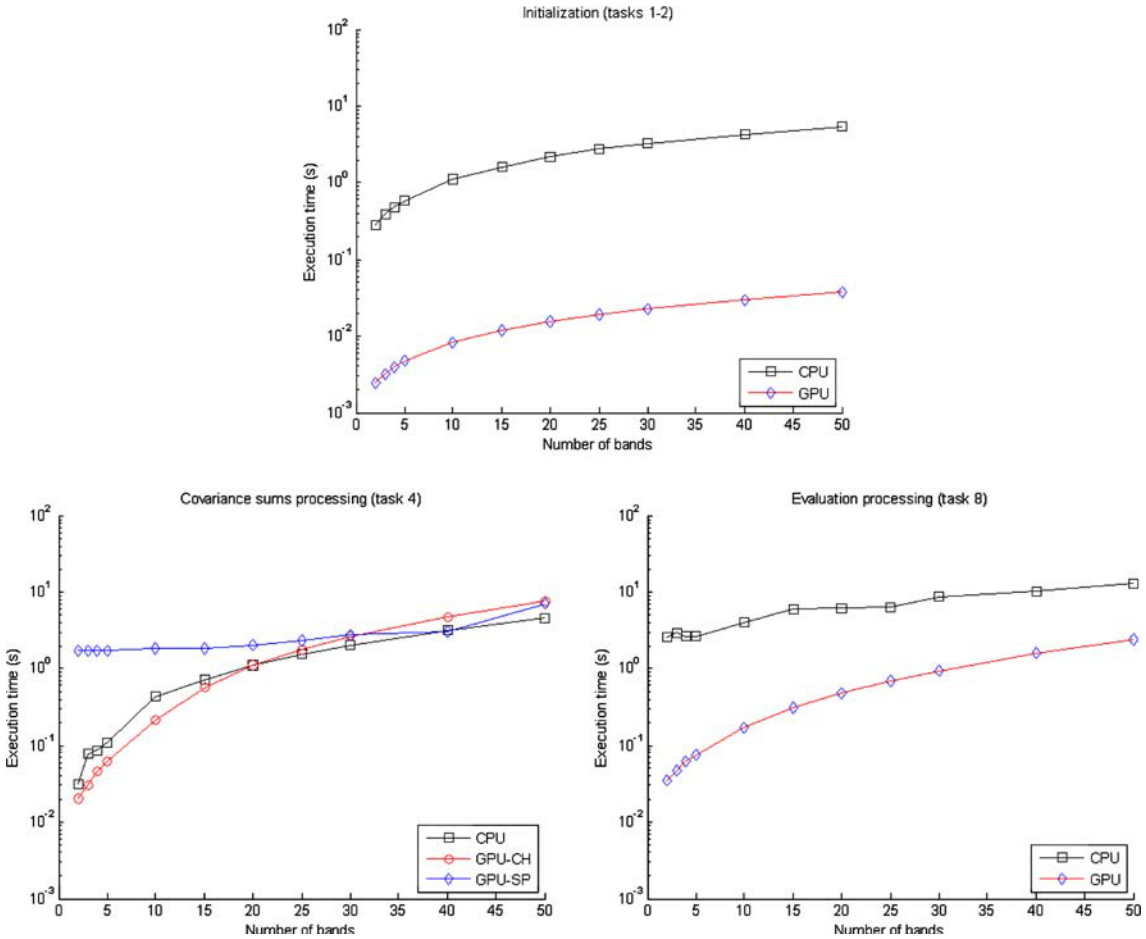


**Fig. 6** Execution time for important parts of the implementations. The *plots* show the difference between the implementations in execution time for the initialization tasks (*top*), the covariance sums processing task (*bottom left*) and the evaluation processing task (*bottom right*). Here, the bands are varied while keeping the other parameters fixed at 10 components, 10 iterations and a subset size of 192,000 pixels

GPU-CH implementation is likely to become more efficient, relative to CPU, for higher number of bands.

The GPU-SP implementation becomes interesting when the number of bands $B > 25$. The reason can be deducted from the algorithm, which explores a spectral-level parallelism. The GPU-SP implementation is faster than both other implementations when $B = 40$. However, for $B = 50$ it is slower than the CPU implementation. The probable reason is that the GPU-code is not well tuned for this problem size.

It is also evident that the gain in GPU-based evaluation processing is decreasing with higher band counts, although it is still significant for 50 bands. Interestingly, the GPU implementations of the initialization part achieves a speed-up gain of around 100. Since most of the initialization corresponds to significant parts of the K-means clustering algorithm, this result also demonstrates that parallel implementations of K-means on GPUs can give a significant increase in computing speed.

### 5.4 Real-time anomaly detection demonstration

After establishing that the parallel GPU implementations are significantly faster than the CPU implementation, we will now demonstrate the impact this has on anomaly detection processing. This experiment will consider the anomaly detector described in Sect. 2 applied in a search and rescue context. While a typical application would process the data in several consecutive blocks, we will here consider the processing of only one such block, and assume that the results obtained are representative for a string of blocks in average over time. Real-time performance is evaluated by comparing the block processing time with the actual time it took to record the block with the hyperspectral camera.

As opposed to the previous experiment, the iterative procedure involved in the estimation process will here stop only when the convergence criterion is satisfied. For this demonstration a convergence threshold of $\delta = 3\%$ was

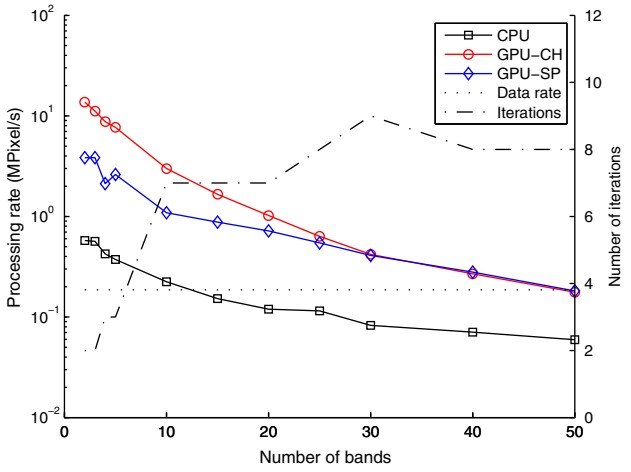


**Fig. 7** Anomaly detection processing rate for different choices of number of bands. The *solid lines* show the processing rate for the different implementations, while the *dotted line* shows the sensor data rate (see the left *y*-axis). The *dash-dot line* shows the number of iterations needed to reach convergence for the different band counts (see the right *y*-axis)

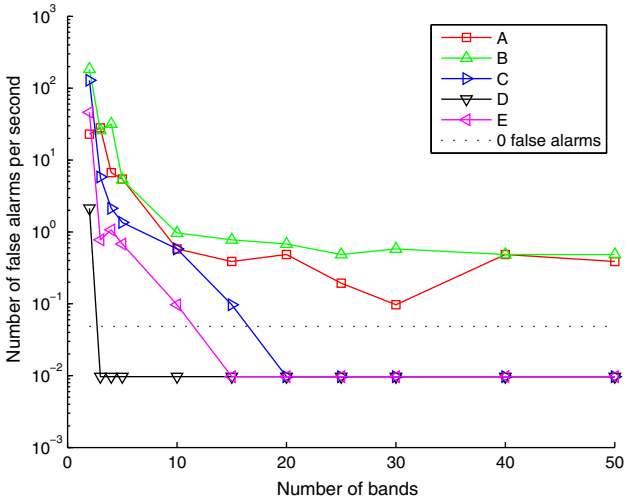


**Fig. 8** Detection results for the GPU-CH approach for different choices of number of bands. Detection results without false alarms are placed below the *dotted line*

chosen. In addition, the number of components was chosen to be $C_{\max} = 10$ and the size of the estimation subset was set to be 10% of the pixels in the image block.

Figure 7 shows the processing rates when each of the three implementations is applied to the different spectrally downsampled images. To be fairly certain that the observed rates are not an extreme result from the random initialization of the estimation, the median rate of 19 runs is chosen for each implementation and band configuration. By comparing with the sensor data rate, represented by the dotted line in Fig. 7, we see that the parallel GPU-based implementations run faster than the data rate right up to about 50 bands. Hence, by exploiting the power of GPU processing, multivariate normal mixture based anomaly detection can be run in real time under similar conditions for less than 50 spectral bands on current hardware. In the 15–25 band interval, the GPU implementations are about 3-10 times faster than the real-time constraint, while the CPU implementation is slower than real time above 10 bands.

When comparing Fig. 7 with Fig. 5 it is clear that the implementations are somewhat faster in this experiment. This is simply because the estimation process needs fewer iterations before satisfying the convergence criterion. Figure 7 also shows the number of iterations needed for the different band configurations.

To fully justify the claim that multivariate normal mixture based anomaly detection is performed in real time, sufficiently good detection results must be demonstrated. The detection results for the GPU-CH implementations are presented in Fig. 8. For 20 bands all the targets are detected with less than 1 false alarm per s, and 3 targets are detected without false alarms. These are considered acceptable results for the target detection scenario in question, and may be further improved by exploiting the available processing time to use more accurate model estimation techniques and perform different false alarm mitigation methods (e.g. [14]). Figure 9 shows the detection result for 20 bands with a detection threshold set so that all targets are detected.

## 6 Conclusion

Multivariate normal mixture models form the basis of an algorithm for anomaly detection in hyperspectral images. The algorithm possesses a significant amount of data-level concurrency in its time-consuming parts, and appears well adapted to the GPU architecture. We have used CUDA to implement the computationally intensive parts of the algorithm on an Nvidia GeForce 8800 GPU, and compared its performance to a CPU-based implementation running on a dual quad-core computer.

Generally, the GPU provides a significant speedup of the algorithm compared to the CPU implementation. The relative performance of the GPU depends on the algorithm parameters such as data size and band count. Furthermore, it is often difficult to optimize GPU code without adapting it to a narrow range of parameters. For the pixel-parallel parts of the algorithm, speedups on the order of 10 and even 100 are observed. For the computation of covariances, however, the GPU only provides an advantage over the CPU for band counts below about 20. For higher band counts, the memory model of the GPU

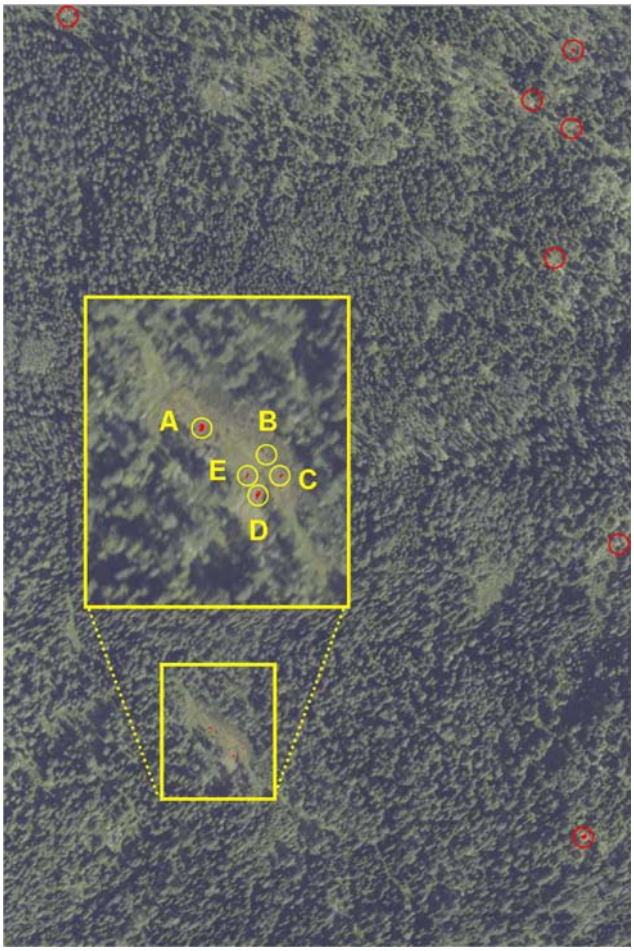


**Fig. 9** Detection results for the GPU-CH approach when detecting all targets using 20 bands. The *red circles* show the false alarms, while the *yellow circles* show the detected targets. The image in the background is a colour image extracted from the hyperspectral image block used in the experiments

does not provide a speed advantage in the calculation of covariance sums.

Anomaly detection has been performed on a realistic hyperspectral data set. We have shown, crucially, that the GPU enables real-time execution of the algorithm on a hyperspectral data stream with high spatial and spectral resolution, with acceptable detection performance and a significant margin on computing time. This margin enables the same hardware to execute other parts of the detection system such as threshold estimation, spatial analysis, false alarm mitigation or signature-based spectral detection.

Finally, it can be noted that methods based on multivariate normal mixtures are versatile statistical tools with potential use in many areas beyond remote sensing. Up to now, computational complexity has precluded their use in many applications. This is about to become history with the advent of highly parallel processing in desktop computers.



## References

1. AMD: ATI CTM Guide (2006)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Croz, J.D., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' Guide (3rd edn). Society for Industrial and Applied Mathematics, Philadelphia (1999)
3. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco (2001)
4. Gonzalez, R.C., Woods, R.E.: Digital Image Processing. Addison-Wesley Publishing Company, Reading (1993)
5. Kåsen, I., Goa, P.E., Skauli, T.: Target detection in hyperspectral images based on multicomponent statistical models for representation of background clutter. In: Proceedings of the SPIE. 5612, pp. 258–264 (2004)
6. Kåsen, I., Rødningsby, A., Haavardsholm, T.V., Skauli, T.: Band selection for hyperspectral target detection based on a multinormal mixture anomaly detection algorithm. In: Proceedings of the SPIE. 6966, p. 696606 (2008)
7. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. **5**(3), 308–323 (2006)
8. Masson, P., Pieczynski, W.: SEM algorithm and unsupervised statistical segmentation of satellite images. IEEE Trans. Geos. and Remote Sens. **31**(3), 618–633 (1993)
9. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley Professional, Boston (2005)
10. Norsk Elektro Optikk AS.: For further information about the HySpex camera: http://www.neo.no/hyspex/ (2008)
11. Nicolescu, C., Jonker, P.: A data and task parallel image processing environment. Parallel Comput. **28**(7–8), 945–965 (2002)
12. NVIDIA: NVIDIA CUDA compute unified device architecture—programming guide. http://developer.nvidia.com/cuda (2007)
13. Plaza, A., Valencia, D., Plaza, J., Martinez, P.: Commodity cluster-based parallel processing of hyperspectral imagery. J. Parallel Distrib. Comput. **66**(3), 345–358 (2006)
14. Schaum, A.: A remedy for nonstationarity in background transition regions for real time hyperspectral detection. In: 2006 IEEE Aerospace Conference, p. 9 (2006)
15. Setoain, J., Tenllado, C., Prieto, M., Valencia, D., Plaza, A., Plaza J.: Parallel hyperspectral image processing on commodity graphics hardware. In: Proceedings of the ICPPW '06, pp. 465–472 (2006)
16. Shen, S.S., Bassett, E.M.: Information-theory-based band selection and utility evaluation for reflective spectral systems. In: Proceedings of the SPIE. 4725, pp. 18–29 (2002)
17. Stein, D.W.J., Beaven, S.G., Hoff, L.E., Winter, E.M., Schaum, A.P., Stocker, A.D.: Anomaly detection from hyperspectral imagery. IEEE Signal Process Mag. **19**(1), 58–69 (2002)
18. Stellman, C.M., Olchowski, F.M., Michalowicz, J.V.: WAR HORSE (Wide Area Reconnaissance—Hyperspectral Overhead Real-time Surveillance Experiment). In: Proceedings of the SPIE. **4379**, pp. 339–346 (2001)

19. Stevenson, B., O'Connor, R., Kendall, W., Stocker, A., Schaff, W., Alexa, D., Salvador, J., Eismann, M. Barnard, K., Kershenstein, J.: Design and performance of the Civil Air Patrol ARCHER hyperspectral processing system. In: Proceedings of the SPIE. **5806**, pp. 731–742 (2005)

## Author Biographies

**Yuliya Tarabalka** has received her Bachelor degree in Computer Science (2005) from the Ternopil Ivan Pul'uj State Technical University, Ukraine and M.Sc. degree in Signal and Image Processing (2007) from National Polytechnic Institute of Grenoble (INPG), France. From July 2007 to January 2008 she worked as a researcher with the Norwegian Defence Research Establishment, Norway. Since 2007, she is a Ph.D. student, pursuing a co-joint degree between University of Iceland, Iceland and INPG, France. Her current research work is funded by the "HYPER-I-NET" Marie Curie Research Training Network. Her research interests are in the areas of image processing, pattern recognition, hyperspectral imaging and development of efficient algorithms.

**Trym Vegard Haavardsholm** received his M.Sc. in Modelling and Data analysis at the University of Oslo, Norway in 2005. He is currently working at the Norwegian Defence Research Establishment as a scientist and Ph.D. student. His main research interests include image analysis and statistical modelling in hyperspectral and ladar images.

**Ingebjørg Kåsen** holds a M.Sc. in mathematics from the University of Oslo, Norway. She has worked as a research scientist at FFI, the Norwegian Defence Research Establishment, since 2002. Research interests include hyperspectral imaging, signal and image processing and statistical modelling.

**Torbjørn Skauli** holds a Ph.D. in physics from the University of Oslo and has worked at FFI, the Norwegian Defence Research Establishment, since 1990. Skauli has worked with semiconductor physics, infrared detectors, electronics, nonlinear optics and nanotechnology. Since 2001 he has been head of the hyperspectral imaging group at FFI.