PARALLEL PROCESSING FOR NORMAL MIXTURE MODELS OF HYPERSPECTRAL DATA USING A GRAPHICS PROCESSOR

Yuliya Tarabalka, Trym Vegard Haavardsholm, Ingebjørg Kåsen, Torbjørn Skauli
Norwegian Defence Research Establishment (FFI), P. O. Box 25, 2027 Kjeller, Norway
e-mail: yultar@gmail.com

ABSTRACT
Multivariate normal mixture models, where a complex statistical distribution is represented by a weighted sum of several multivariate normal probability distributions, have many potential applications including anomaly detection (AD) in hyperspectral (HS) images. The high computational cost of mixture models requires hardware and/or algorithmic acceleration to make AD run in real time. In this paper we describe the concurrency present in the AD algorithm that includes a normal mixture estimation task. We explore the use of graphics processing units (GPUs) for parallel implementation of the algorithm. The GPU implementations provide a significant speedup compared to multi-core central processing unit (CPU) implementations, and enable the algorithm to execute in real time.

Index Terms— hyperspectral image, anomaly detection, multivariate normal mixture model, GPU processing

1. INTRODUCTION
Hyperspectral imaging is a powerful technique capable of discerning different physical substances based on their spectral properties. HS sensors typically capture tens or hundreds of spectral bands for each spatial position in the image, so that each pixel in a HS image is presented as a vector of values corresponding to the detailed spectrum of the received light. One possible application is detection of targets with low visual contrast.

In remote sensing applications, statistical methods are an attractive approach for processing of HS data. However, common distribution functions such as the multinormal distribution are often not able to represent well the distributions occurring in practice such as a background distribution in the case of target detection. For this, mixture models offer a solution where an arbitrary probability density function (PDF) is approximated by a finite sum of simple PDFs [1].

In this paper we study the use of normal mixture models (NMM) to represent the variability of an image background, in order to perform anomaly detection. Many applications of anomaly detection must be executed in real-time, such as military or rescue operations. Thus high-performance implementations must be developed. We explore the use of parallel processing on GPUs for target detection in HS data. GPUs offer very high computing performance for tasks that can be parallelized and adapted to the GPU architecture. Ongoing development by major manufacturers is rapidly making GPUs more accessible for general-purpose computation through adaptation of processor architectures and availability of software development tools [2]. The mass market for computer games is currently driving the GPUs to evolve faster than conventional processors in terms of performance while keeping the cost and power consumption fairly low. We compare the performance of optimized codes running on a GPU and a high-end multi-core CPU for a prototypical test task representative of spectral anomaly target detection.

2. ANOMALY DETECTION ALGORITHM
The method for anomaly detection based on multi-component statistical models considered in this paper is outlined in [3].

A HS image can be considered as a set of pixel vectors \( X = \{x_j \in \mathbb{R}^B, j = 1, 2, ..., n \} \), where \( n \) is a number of image pixels and \( B \) is a number of spectral bands. A \( C \)-component mixture of normals is represented by the probability density function:

\[
p(x) = \sum_{c=1}^{C} \omega_c N_c(x; \mu_c, \Sigma_c)
\]

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

\[
N_c(x; \mu_c, \Sigma_c) = \frac{1}{(2\pi)^{B/2} |\Sigma_c|^{1/2}} \exp\left\{-\frac{1}{2}(x - \mu_c)^T \Sigma_c^{-1} (x - \mu_c) \right\}.
\]

The NMM for the distribution of pixel spectra is fitted iteratively using a method similar to the SEM algorithm [4].

We do not use the whole HS image for parameters estimation, as it can be too time-consuming. Instead, we construct a subset of \( m \) pixel vectors \( S = \{s_j \in \mathbb{R}^B, j = 1, 2, ..., m \} \), \( S \subseteq X \), taking each \( n/m \)-th pixel of the image \( X \).
The principal idea of the NMM estimation algorithm (Algorithm 1 below) is to assume that each pixel \( s_j \) from \( S \) belongs to one of the components \( c = 1, 2, ..., C \). Thus, on each iteration \( i \) we obtain a partition \( Q^i_c \), \( c = 1, 2, ..., C \) of the subset \( S \), where \( Q^i_c = \{ s_{j,c} \in R^B, j = 1, 2, ..., m^i_c \} \) contains pixels belonging to the component \( c \) on the iteration \( i \).

**Algorithm 1 Normal mixture model estimation**

**Require:**
- subset of HS image pixels \( S \)
- parameters: upper bound on the number of components \( C \), threshold \( \delta \) for termination of the iteration process, maximum number of iterations \( I_{max} \)

**Output:** \( C, \omega_c, \mu_c, \Sigma_c \) for \( c = 1, 2, ..., C \)

**Initialization (Iteration 0):**
Determine the first partition \( Q^0_c, c = 1, 2, ..., C \) of \( S \).
1. Choose randomly \( C \) pixels from the subset \( S \) to serve as component (cluster) centers.
2. Assign pixels of the subset \( S \) to the components on the basis of the nearest Euclidean distance to the component center.

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

**Parameter estimation step:**
Estimate \( \mu^i_c, \Sigma^i_c \) and \( \omega^i_c \) for \( c = 1, 2, ..., C \) using the component-wise Maximum Likelihood estimates:

\[
\mu^i_c = \frac{1}{m^i_c} \sum_{j=1}^{m^i_c} x^i_{j,c} \tag{3}
\]

\[
\Sigma^i_c = \frac{1}{m^i_c} \sum_{j=1}^{m^i_c} (x^i_{j,c} - \mu^i_c)(x^i_{j,c} - \mu^i_c)^T \tag{4}
\]

\[
\omega^i_c = \frac{m^i_c}{m}. \tag{5}
\]

**Component assignment step:**
1. Assign each pixel to one of the components, using the maximum a posteriori probability criteria:

\[
x_j \in Q^i_c : \ln(\omega^i_cN_c(\mu^i_c, \Sigma^i_c; x_j)) = \max_{l}(\ln(\omega^i_lN_l(\mu^i_l, \Sigma^i_l; x_j))). \tag{6}
\]

2. If \( m^i_c \leq B, c = 1, 2, ..., C \), eliminate component \( C \). Next, if at least one component is eliminated, return to the parameter estimation step.

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

Having estimated the NMM for the background, a probability value for each pixel in the set \( X \) is calculated according to the background model by using formula 1.

The overall computational complexity for the NMM estimation part of the AD algorithm is \( O(mCB^2I) \), and for the probability map calculation part it is \( O(nCB^2) \). Table 1 summarizes the complexities for the AD algorithm parts with the highest computational cost.

Plaza et al. [5] have described some perspectives of parallel processing on HS images. In the next subsection we will show a significant potential of parallel implementation of the proposed AD algorithm.

### 2.1. Finding concurrency in the AD algorithm

Setoain et al. [2] distinguish task-level, spatial-level and spectral-level parallelism for HS image processing algorithms.

All four algorithm parts from Table 1 can be parallelized. The parts that assign pixels to the components, and the probability map calculation task exhibit inherent parallelism at pixel level. The component membership and the background probability value for each pixel can be calculated independently on several processing elements (PEs). This results in a
The number of tasks that can be run concurrently is equal to the number of pixels in the set. When the values of \( n, m \) are high, the amount of concurrency (level of scalability) is significant.

A more challenging step is the calculation of what we here call the covariance sums: \( Z'_{s,c} = m_c^{-1} \Sigma_c^{-1} \). Here \( CB(B+1)/2 \) elements must be estimated. We consider two approaches to parallelize this task.

### 2.1.2. Covariance sum - spectral-level parallelism (SP)

Another way to parallelize the covariance sums estimation is to calculate the covariance sums for all the parts in parallel, so that each task will estimate covariance sums for one chunk of the subset. The next step consists in calculating covariance sums for the whole subset, summing in parallel covariance sums for its parts (see algorithm 2).

#### Algorithm 2 Covariance sum - chunking approach

1. Decompose the subset \( S \) into \( K \) parts \( U_k, k = 1, 2, ..., K \) and execute in parallel \( T = K \) tasks:
   
   ```
   for each task \( k \):
     for each pixel \( x_{i,c}^{-1} \in U_k \):
       \( z_{j,c}^{-1} = x_{j,c}^{-1} - \mu_c \);
       \( Z_{c}^i = Z_{c}^i + (z_{j,c}^{-1})(z_{j,c}^{-1})^T \);
     end for
   end for
   ``

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

### 2.1.2. Covariance sum - spectral-level parallelism (SP)

Another way to parallelize the covariance sums estimation is to calculate in parallel the covariance between bands \( q \) and \( r \) \((Z(q,r), q = 1, 2, ..., B; r = 1, ..., q)\), so that each task will calculate \( C \) elements \( Z_{c}(q,r) \), \( c = 1, 2, ..., C \). The algorithm consists of two branching steps: centering of the input subset \( S \) and covariance sums calculation (see algorithm 3).

#### Algorithm 3 Covariance sum - spectral-level parallelism

1. for each pixel \( x_{i,c}^{-1} \in S \) (\( T = m \) concurrent tasks):
   
   ```
   z_{j,c}^{-1} = x_{j,c}^{-1} - \mu_c ;
   ```

2. Each concurrent task (in total \( T = B(B+1)/2 \) tasks) calculates \( Z_{c}(q,r), c = 1, 2, ..., C \).

### 3. GPU-BASED PARALLEL IMPLEMENTATION

The potential advantages of using graphics hardware for HS image processing have been explored by Setoain et al. [2].

For the experiments presented here we have chosen to implement the parallel AD algorithms on an NVidia GeForce 8800 Ultra GPU, exploiting the new CUDA (Compute Unified Device Architecture) technology [6]. The CUDA hardware and software architecture simplifies the use of GPUs as massively parallel processing computing devices. The GPU’s capability to execute a very high number of threads in parallel, with fast access to a shared memory between threads, makes it appealing to use for implementation of the described AD algorithms. The most computationally demanding parts of the algorithm have been implemented into seven GPU functions (kernels) that are summarized in Table 1.

### 4. EXPERIMENTAL RESULTS

Our experiments were performed on a HP xw8400 Workstation running Windows XP32 on a dual Quad-Core Intel Xeon CPU E5345 (2.33 GHz, 1333 MHz system bus), 3 GB RAM and NVidia GeForce 8800 Ultra GPU. Three different implementations of the test task have been made, one for running on the CPU only, and two alternative GPU-based implementations. The CPU implementation is built with the Intel C++ Compiler 9.1 using OMP, BLAS and LAPACK libraries, while the GPU implementations have been made using the CUDA compiler driver nvcc [6].

The data used for the experiments are airborne recordings of a forest scene in Norway made by the HySpex HS camera [7]. The image is a 1600 x 1200 pixels block of the original recording. To evaluate the scalability of implementations, we used spectrally downsampled blocks with 2 to 50 bands, obtained by averaging over adjacent bands.

We evaluate the performance of the implementations by measuring the program 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 \( S \) and the number of iterations \( I \).

Figures 1 and 2 show the total execution time for the test task implemented on the CPU and the GPU. Separate timings are shown in Figure 1 for the final assignment of probabilities and for the total time spent on calculation of covariance sums.

The overall result is that the GPU increases computing speed by a significant factor. From Figure 1 we see that 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 matrix dimensionality. When parameters \( C, m \) or \( I \) are varied (ref: Figure 2) and the number of
bands is fixed to $B = 15$, the GPU-chunking implementation provides the highest performance, with the speedup versus CPU implementation of order 10. An important observation is that the speedup brought by the GPU makes the processing faster than the data rate of many relevant sensors, and hence enables real time processing based on mixture models.

5. CONCLUSIONS

In this work, we have explored the use of parallel processing on graphics processors for normal mixture model estimation, using anomaly detection in hyperspectral data as a test task. We have shown that the studied task possesses a high degree of concurrency, and that its implementation on GPU provides significant speedups. The adaptation to GPU processing enables the anomaly detection algorithm to run in real time on airborne HS data. The results may serve as a benchmark for currently achievable performance with normal mixture models as well as normal distributions in a range of applications beyond hyperspectral imaging.

6. REFERENCES