Performance Portability Study of an Automatic Target Detection and Classification Algorithm for Hyperspectral Image Analysis using OpenCL

Sergio Bernabe[1], Francisco D. Igual[1], Guillermo Botella[1], Carlos Garcia[1], Manuel Prieto-Matias[1] and Antonio Plaza[2]

[1]Complutense University, Madrid, Spain
[2]Hyperspectral Computing Laboratory, University of Extremadura, Caceres, Spain

ABSTRACT

Recent advances in heterogeneous high performance computing (HPC) have opened new avenues for demanding remote sensing applications. Perhaps one of the most popular algorithm in target detection and identification is the automatic target detection and classification algorithm (ATDCA) widely used in the hyperspectral image analysis community. Previous research has already investigated the mapping of ATDCA on graphics processing units (GPUs) and field programmable gate arrays (FPGAs), showing impressive speedup factors that allow its exploitation in time-critical scenarios. Based on these studies, our work explores the performance portability of a tuned OpenCL implementation across a range of processing devices including multicore processors, GPUs and other accelerators. This approach differs from previous papers, which focused on achieving the optimal performance on each platform. Here, we are more interested in the following issues: (1) evaluating if a single code written in OpenCL allows us to achieve acceptable performance across all of them, and (2) assessing the gap between our portable OpenCL code and those hand-tuned versions previously investigated. Our study includes the analysis of different tuning techniques that expose data parallelism as well as enable an efficient exploitation of the complex memory hierarchies found in these new heterogeneous devices.

Experiments have been conducted using hyperspectral data sets collected by NASA's Airborne Visible Infrared Imaging Spectrometer (AVIRIS) and the Hyperspectral Digital Imagery Collection Experiment (HYDICE) sensors. To the best of our knowledge, this kind of analysis has not been previously conducted in the hyperspectral imaging processing literature, and in our opinion it is very important in order to really calibrate the possibility of using heterogeneous platforms for efficient hyperspectral imaging processing in real remote sensing missions.

Keywords: Hyperspectral imaging, automatic target detection and classification algorithm (ATDCA), high performance computing (HPC), OpenCL, performance portability

1. INTRODUCTION

In recent years, many hyperspectral image analysis algorithms have been mapped in heterogeneous high performance computing for demanding remote sensing applications,[1] such as target detection and identification for military purposes, biological threat detection, monitoring of oil spills and other types of chemical contamination, wildfire tracking, and so on.

Previous research studies[2] have shown that the ever-growing computational demands of these applications, which often require real- or near real-time responses, can fully benefit from these emerging computing platforms. Unfortunately, programming heterogeneous systems is a laborious task that often involves a deep knowledge of the underlying architecture. In fact, programmers are usually forced to concentrate on implementation details and learning new APIs rather than on other important issues related to the application. Furthermore, the lack of standardization in the first generation products from IBM (Cell BE) or NVIDIA resulted in most applications being too specific to a given architecture, eliminating or at least making extremely difficult the possibility of reusing code across different platforms. Many proprietary standards and tools have been designed in order to cover a closed set of architectures, and OpenCL has become a free standard for parallel programming on heterogeneous systems.

High-Performance Computing in Remote Sensing V, edited by Bormin Huang, Sebastián López, Zhensen Wu, Jose M. Nascimento, Boris A. Alpatov, Jordi Portell de Mora, Proc. of SPIE Vol. 9646, 96460M
 · doi: 10.1117/12.2195102

The necessity to improve portability[3] led to the development of OpenCL, an open and royalty-free standard based on C99 for parallel programming on heterogeneous systems. Its first specification was released in late 2008. Since then, it has been adopted by many vendors for all sort of computing devices, from dense multicore systems to new accelerators such as graphics processing units (GPUs), digital signal processors (DSPs), field programmable gate arrays (FPGAs), the Intel Xeon-Phi and other custom devices. The main advantage of implementing OpenCL concerns the shorter time to market and faster implementations. However, OpenCL makes no guarantee of performance portability. Portability is important for application and library developers, but performance is what really matters to application users. Our focus in this work is on analyzing and studying this emerging problem using as a benchmark the automatic target detection and classification algorithm (ATDCA),[4] a well-known algorithm that has been widely used for the detection of (moving or static) targets in remotely sensed hyperspectral images.

Previous research has already investigated the mapping of ATDCA using the Gram-Schmidt method for orthogonalization on GPUs[5] and designs on FPGAs,[6] showing impressive speedup factors that allow its exploitation in time-critical scenarios. Based on these studies, our work explores the performance portability of a tuned OpenCL implementation across a range of processing devices including multicore processors, GPUs and accelerators. This approach differs from previous papers, which focused on achieving the optimal performance on each platform. In this paper, we are more interested in the following issues: (1) evaluating if a single code written in OpenCL allows us to achieve acceptable performance across all of them, and (2) assessing the gap between our portable OpenCL code and those hand-tuned versions previously investigated. Our study includes the analysis of different tuning techniques that expose data parallelism using OpenMP (open multi-processing) and CUDA (compute unified device architecture) as well as enable an efficient exploitation of the complex memory hierarchies found in these new heterogeneous devices.

The experimental results have been conducted using hyperspectral data sets collected by NASA's Airborne Visible Infra-red Imaging Spectrometer (AVIRIS)[7] and the Hyperspectral Digital Imagery Collection Experiment (HYDICE) sensors. To the best of our knowledge, this kind of analysis has not been previously conducted in the hyperspectral imaging processing literature, and in our opinion it is very important in order to really calibrate the possibility of using heterogeneous platforms for efficient hyperspectral imaging processing in real remote sensing missions.

The remainder of this paper is organized as follows. Section 2 describes the optimized ATDCA method. Section 3 describes the proposed parallel implementation. Section 4 presents an experimental evaluation of the proposed implementation in terms of both accuracy and parallel performance using real data sets on heterogeneous platforms. Finally, Section 5 presents as few concluding remarks and pointers to future work.

2. THE OPTIMIZED ATDCA ALGORITHM

The original ATDCA algorithm was originaly developed in[8] to find spectrally distinct signatures using orthogonal subspace projections (OSP). For this work, we have used an optimization of this algorithm (see[5,9]) which allows calculating orthogonal projections without requiring the computation of the inverse of the matrix that contains the targets already identified in the image. This operation, which is difficult to implement in parallel, is accomplished using the Gram-Schmidt (GS) method for orthogonalization. This process selects a finite set of linearly independent vectors $\mathbf{A} = \{\mathbf{a}_1, \cdots, \mathbf{a}_p\}$ in the inner product space $\mathbf{R}^L$ in which the original hyperspectral image is defined, and generates an orthogonal set of vectors $\mathbf{B} = \{\mathbf{b}_1, \cdots, \mathbf{b}_p\}$ which spans the same *p-dimensional* subspace of $\mathbf{R}^L$ ($p \leq L$) as $\mathbf{A}$. In particular, $\mathbf{B}$ is obtained as follows:

$$\begin{array}{ll}
\mathbf{b}_1 = \mathbf{a}_1, & \mathbf{e}_1 = \frac{\mathbf{b}_1}{\|\mathbf{b}_1\|} \\
\mathbf{b}_2 = \mathbf{a}_2 - proj_{\mathbf{b}_1}(\mathbf{a}_2), & \mathbf{e}_2 = \frac{\mathbf{b}_2}{\|\mathbf{b}_2\|} \\
\mathbf{b}_3 = \mathbf{a}_3 - proj_{\mathbf{b}_1}(\mathbf{a}_3) - proj_{\mathbf{b}_2}(\mathbf{a}_3), & \mathbf{e}_3 = \frac{\mathbf{b}_3}{\|\mathbf{b}_3\|} \\
\mathbf{b}_4 = \mathbf{a}_4 - proj_{\mathbf{b}_1}(\mathbf{a}_4) - proj_{\mathbf{b}_2}(\mathbf{a}_4) - proj_{\mathbf{b}_3}(\mathbf{a}_4), & \mathbf{e}_4 = \frac{\mathbf{b}_4}{\|\mathbf{b}_4\|} \\
\vdots & \vdots \\
\mathbf{b}_p = \mathbf{a}_p - \sum_{j=1}^{p-1} proj_{\mathbf{b}_j}(\mathbf{a}_p), & \mathbf{e}_p = \frac{\mathbf{b}_p}{\|\mathbf{b}_p\|},
\end{array} \tag{1}$$

where the projection operator is defined in (2), in which $< \mathbf{a}, \mathbf{b} >$ denotes the inner product of vectors $\mathbf{a}$ and $\mathbf{b}$.

$$proj_{\mathbf{b}}(\mathbf{a}) = \frac{< \mathbf{a}, \mathbf{b} >}{< \mathbf{b}, \mathbf{b} >} \mathbf{b}. \quad (2)$$

The sequence $\mathbf{b}_1, \cdots, \mathbf{b}_p$ in (1) represents the set of orthogonal vectors generated by the Gram-Schmidt method, and thus, the normalized vectors $\mathbf{e}_1, \cdots, \mathbf{e}_p$ in (1) form an orthonormal set. As far as $\mathbf{B}$ spans the same p-dimensional subspace of $\mathbf{R}^L$ as $\mathbf{A}$, an additional vector $\mathbf{b}_{p+1}$ computed by following the procedure stated at (1) is also orthogonal to all the vectors included in $\mathbf{A}$ and $\mathbf{B}$. This algebraic assertion constitutes the cornerstone of the ATDCA method with Gram-Schmidt orthogonalization, referred to hereinafter as ATDCA-GS algorithm, whose pseudocode is represented in Algorithm 1 (see[5] for more details).

Algorithm 1 Pseudocode of ATDCA-GS

INPUTS: $\mathbf{F} \in \mathbf{R}^n$ and t;
% $\mathbf{F}$ denotes an n-dimensional hyperspectral image with r pixels and t denotes the number of targets to be detected
$\mathbf{U} = [\mathbf{x}_0 \mid 0 \mid, \cdots, \mid 0]$;
% $\mathbf{x}_0$ is the pixel vector with maximum length in $\mathbf{F}$
$\mathbf{B} = [0 \mid 0 \mid, \cdots, \mid 0 \mid]$;
% $\mathbf{B}$ is an auxiliary matrix for storing the orthogonal base generated by the Gram-Schmidt process
for $i = 1$ to $t - 1$ **do**
  $\mathbf{B}[:, i] = \mathbf{U}[:, i]$;
  % the i-th column of $\mathbf{B}$ is initialized with the target computed in the last iteration (here, the operator ":" denotes "all elements")
  $P^{\perp}_{\mathbf{U}} = [\mathbf{1}, \cdots, \mathbf{1}]$;
  for $j = 2$ to i **do**
    $proj_{\mathbf{B}[:,j-1]}(\mathbf{U}[:, i]) = \frac{\mathbf{U}[:,i]^T \mathbf{B}[:,j-1]}{\mathbf{B}[:,j-1]^T \mathbf{B}[:,j-1]} \mathbf{B}[:, j-1]$;
    $\mathbf{B}[:, i] = \mathbf{B}[:, i] - proj_{\mathbf{B}[:,j-1]}(\mathbf{U}[:, i])$;
    % The i-th column of $\mathbf{B}$ is updated
  end for j
  % The computation of $\mathbf{B}$ is finished for the current iteration of the main loop
  for $k = 1$ to i **do**
    $proj_{\mathbf{B}[:,k]}(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{B}[:,k]}{\mathbf{B}[:,k]^T \mathbf{B}[:,k]} \mathbf{B}[:, k]$;
    $P^{\perp}_{\mathbf{U}} = P^{\perp}_{\mathbf{U}} - proj_{\mathbf{B}[:,k]}(\mathbf{w})$;
  end for k
  % The computation of $P^{\perp}_{\mathbf{U}}$ is finished for the current iteration of the main loop
  $\mathbf{v} = P^{\perp}_{\mathbf{U}} \mathbf{F}$;
  % $\mathbf{F}$ is projected onto the direction indicated by $P^{\perp}_{\mathbf{U}}$
  $i = \text{argmax}_{\{1,\cdots,r\}} \mathbf{v}[:, i]$;
  % The maximum projection value is found, where r denotes the total number of pixels in the hyperspectral image
  $\mathbf{x}_i \equiv \mathbf{U}[:, i+1] = \mathbf{F}[:, i]$;
  % The target matrix is updated
end for
OUTPUT: $\mathbf{U} = [\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{t-1}]$;
% $\mathbf{U}$ denotes the matrix with all the spectral signatures for each target

3. OPENCL FRAMEWORK AND IMPLEMENTATIONS

In this section, we show our mapping of the ATDCA algorithm using a new portable OpenCL code for a diverse set of heterogeneous platforms. In addition, the OpenMP and CUDA implementations will be explained in the next subsections.

3.1 OpenCL framework

The OpenCL* specification defines a platform, memory and programming model which permits many add-ons that are vendor specific, cross-vendor and Khronos. There is considerable freedom in terms of implementing the

*https://www.khronos.org/opencl

platform providing the final implementation satisfies the OpenCL specifications. In OpenCL, the CPU and its memory are denoted as the *host* and the parallel platform and its memory as the *device*. The code executed in parallel using the heterogeneous platform is typically called *kernel* and can have parameters as C language functions.

An OpenCL kernel executes in parallel across a set of parallel threads named *work-items*. A *work-group* is a set of concurrent work-items that can cooperate among themselves. An index space is a set of work-groups that may be executed independently and may thus execute in parallel. The dimensions of the index space and work-groups are set by the programmer in the call to the kernel and depend on the specific model of the platform in use (see Section 4).

Algorithm 2 OpenCL Brightest_pixel_spectra Kernel

```
1: global d_image ← Initial F vector
2: global d_bright ← The bright value for each pixel spectral x_i in F
3: registers bright ← 0, value ← 0
4: id ← get_group_id(0) * get_local_size(0) + get_local_id(0)
   % n_b denotes the number of spectral bands
5: if id < r then
6:    for k = 0 to n_b do
7:       value ← d_image[id + (k * r)]
8:       bright ← bright + value * value
9:    end for
10:   d_bright[id] ← bright
11: end if
```

Algorithm 3 OpenCL Pixel_projection Kernel

```
1: global d_image ← Initial F vector
2: global d_projection ← The projection value for each pixel spectral x_i in F
3: global d_f ← The most orthogonal vector
4: registers sum ← 0, value ← 0
5: local s_df[n_b] ← Initial d_f structure with the most orthogonal vector
6: id ← get_group_id(0) * get_local_size(0) + get_local_id(0)
7: if id < r then
8:    if get_local_size(0) < n_b then
9:       for i = get_local_id(0) to n_b do
10:         s_df[i] ← d_f[i]
11:      end for
12:   else
13:      if get_local_id(0) < n_b then
14:         s_df[get_local_id(0)] ← d_f[get_local_id(0)]
15:      end if
16:   end if
17:   barrier(CLK_LOCAL_MEM_FENCE) % In this barrier, all work-items must wait the execution of all work-items
      in a work-group to complete the copy in local memory of d_f
18:   for i = 0 to n_b do
19:      value ← d_image[id + (i * r)]
20:      sum ← sum + value * s_df[i]
21:   end for
22:   d_projection[id] ← sum * sum
23: end if
```

3.2 OpenCL implementation

By observing the program flow in Algorithm 1 it is possible to identify mainly the potential bottleneck in the ATDCA-GS algorithm. The most important one is the projection of the orthogonal vector onto each pixel spectra

on the image. Once the hyperspectral image $\boldsymbol{F}$ is mapped onto global memory's device, while setting the number of work-groups to be equal to the number of pixel spectra in the hyperspectral image divided by the number of work-items per block, this number will depend on the considered device architecture. A kernel is now used to calculate the brightest pixel spectra $\mathbf{x}_0$ in $\mathbf{F}$. This kernel is outlined in Algorithm 2.

Once the brightest pixel in $\mathbf{F}$ has been identified, the pixel spectra is allocated as the first column in matrix $\mathbf{U}$. To calculate the orthogonal vectors through the GS method, it will be performed in the CPU because this method operates on a small data structure and the results can be obtained very quickly and the transfer time for such a small matrix is also negligible. Then, a kernel is created to project the orthogonal vector onto each pixel in the image. The kernel is outlined in Algorithm 3, where we use the local memory to store the most orthogonal vectors obtained at each iteration of ATDCA-GS. For this purpose, the operation will be much faster and with fewer memory accesses as compared to the case in which these vectors are stored in the global device memory.

Algorithm 4 OpenCL Reduction_projection Kernel

```
1: global d_projection ← The projection value for each pixel spectral x_i in F
2: global d_red_projection ← Initial structure to store the partial reductions
3: global d_index ← The index for each projection value
4: local s_p[BLK] ← Initial structure to store all the projections
5: local s_i[BLK] ← Initial structure to store all the index for each projection
6: tid ← get_local_id(0)
7: id ← get_group_id(0) * get_local_size(0)*2 + get_local_id(0)
8: if id + get_local_size(0) ≥ r then
9:    s_p[tid] ← d_projection[id]
10:   s_i[tid] ← id
11: else
12:   if d_projection[id] > d_projection[id + get_local_size(0)] then
13:      s_p[tid] ← d_projection[id]
14:      s_i[tid] ← id
15:   else
16:      s_p[tid] ← d_projection[id + get_local_size(0)]
17:      s_i[tid] ← (id + get_local_size(0))
18:   end if
19: end if
20: barrier(CLK_LOCAL_MEM_FENCE)
21: for s = get_local_size(0)/2 to s >>= 1 do
22:   if tid < s then
23:      if s_p[tid] ≤ s_p[tid + s] then
24:         s_p[tid] ← s_p[tid + s]
25:         s_i[tid] ← s_i[tid + s]
26:      end if
27:   end if
28:   barrier(CLK_LOCAL_MEM_FENCE)
29: end for
30: d_red_projection[get_group_id(0] ← s_p[0]
31: d_index[get_group_id(0] ← s_i[0]
32: barrier(CLK_LOCAL_MEM_FENCE)
```

The maximum of all projected pixels is calculated using a separate reduction kernel which also uses local memory to store each of the projections and obtains the new target $\mathbf{x}_1$. The kernel is outlined in Algorithm 4. The algorithm 1 now extends the target matrix as $\mathbf{U} = [\mathbf{x}_0\mathbf{x}_1]$ and repeats the same process until the desired number of targets (specified by the input parameter t) has been detected. The output of the algorithm 1 is a set of targets $\mathbf{U} = [\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{t-1}]$.

3.3 CUDA implementation

On the other hand, our implementation in CUDA of ATDCA-GS algorithm for GPUs uses the same strategy performed in.[5] In this case, we have selected the last version of CUDA, version 7.0. For implementation purposes,

Algorithm 5 CUDA Pixel_projection Kernel

global $d\_image \leftarrow$ Initial $\mathbf{F}$ vector $[r]$
global $d\_projection \leftarrow$ The projection value for each pixel spectral $\mathbf{x}_i$ in $\mathbf{F}$
global $d\_f \leftarrow$ The most orthogonal vector
registers sum $\leftarrow$ 0, value $\leftarrow$ 0
shared $s\_df[n_b] \leftarrow$ Initial **d_f** structure with the most orthogonal vector
idx $\leftarrow$ blocDim.x * blockIdx.x + threadIdx.x
if $id < r$ **then**
  if $threadIdx.x < n_b$ **then**
    for $i = threadIdx.x$ to n_b **do**
      $s\_df[i] \leftarrow d\_f[i]$
    end for
  else
    if $threadIdx.x < n_b$ **then**
      $s\_df[threadIdx.x] \leftarrow d\_f[threadIdx.x]$
    end if
  end if
  $\_\_syncthreads()$
  % In this synchronize, all threads must wait the execution of all threads in a block to complete the copy in local memory of d_f
  for $i = 0$ to n_b **do**
    value $\leftarrow$ $d\_image[idx + (i * r)]$
    sum $\leftarrow$ sum + value * $s\_df[i]$
  end for
  $d\_projection[idx] \leftarrow$ sum * sum
end if

Algorithm 6 OpenMP Pixel_projection function

global $h\_image \leftarrow$ Initial $\mathbf{F}$ vector $[r]$
global $h\_f \leftarrow$ The most orthogonal vector
global $max\_local \leftarrow$ 0, $max\_locals[MAX\_THREADS] \leftarrow$ 0, $pos\_abs\_locals[MAX\_THREADS] \leftarrow$ 0
% MAX_THREADS denotes the maximum number to execute the code
registers value $\leftarrow$ 0, value_out $\leftarrow$ 0
$\#pragma$ omp parallel for private(value, value_out, j)
for $iter = 0$ to r **do**
  th $\leftarrow$ omp_get_thread_num()
  for $j = 0$ to n_b **do**
    value $\leftarrow$ value + $h\_image[iter + (j * n_p)] * h\_f[j]$
  end for
  value_out $\leftarrow$ value * value
  if value_out > max_locals[th] **then**
    max_locals[th] $\leftarrow$ value_out
    pos_abs_locals[th] $\leftarrow$ iter
  end if
end for
for $iter = 0$ to MAX_THREADS **do**
  if max_locals[iter] > max_local **then**
    max_local $\leftarrow$ max_locals[iter]
    pos_abs $\leftarrow$ pos_abs_locals[iter]
  end if
end for

the kernel most important is shown in Algorithm 5. In this kernel, we have a difference with the implementation in.[5] The shared memory array is declared by means of an unsized extern array sintax, `__extern__ float s_df[]`. The size is simplicitly determined from the third execution configuration parameter when the kernel is launched. This impact avoid to declare a variable with the size of this array allowing to set the size depending on the input image.

3.4 OpenMP implementation

Finally, our implementation in OpenMP of ATDCA-GS algorithm uses a modified strategy performed in.[9] The potential bottleneck in this implementation is the *pixel_projection* function, where the reduction process is included. In this work, this function is calculated without use locking routines (*omp_set_lock*) and (*omp_unset_lock*) between lines 12-15. This function is outlined in Algorithm 6. The impact of this modification involves an improvement in the scalability of the algorithm.

4. EXPERIMENTAL RESULTS

The experiments are carried out using two hyperspectral images. The first data set used in our experiments was collected by the Hyperspectral Digital Imagery Collection Experiment (HYDICE) sensor was used in experiments, which represents a subset of the well-known forest radiance data consisting of 64 $\times$ 64 pixels and 169 spectral bands for a total size of 5.28 MB. A second hyperspectral image scene has been considered for experiments. It is the well-known AVIRIS Cuprite scene, collected by the Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) in the summer of 1997 and available online in reflectance units after atmospheric correction. The portion used in experiments corresponds to a 350 $\times$ 350-pixels, which comprises 188 spectral bands in the range from 400 to 2500 nm and a total size of around 50 MB. Water absorption bands as well as bands with low signal-to-noise ratio (SNR) were removed prior to the analysis.

To carry out the tests on OpenCL, we have used a multicore heterogeneous system equipped with: two Intel Xeon E5-2670 processors with 8 cores each, at 2.60 GHz and 64 GB of DDR3 RAM memory. An NVIDIA K-20c GPU with features 2496 cores operating at 706 MHz and dedicated memory of 5 GB. And finally, the Xeon-Phi 31S1P coprocessor has 57 cores supporting the execution of four hardware threads (228 hardware threads in total) operating at 1.100 GHz and 8 GB installed RAM memory.

4.1 Accuracy Evaluation

In order to analyze the accuracy of the parallel implementation, the well-known spectral angle distance (SAD)[10] is adopted. For this purpose, we have selected the AVIRIS Cuprite image, where the numer of targets to be extracted was estimated as $t = 19$ after calculating the virtual dimensionality (VD).[11] The SAD was performed between the extracted targets and ground-truth spectral signatures. The results present very low SAD scores under 6 degrees on the average, which indicates that the implementation provides accurate results, since the best case of SAD is 0 degrees and the worst case is 90 degrees.

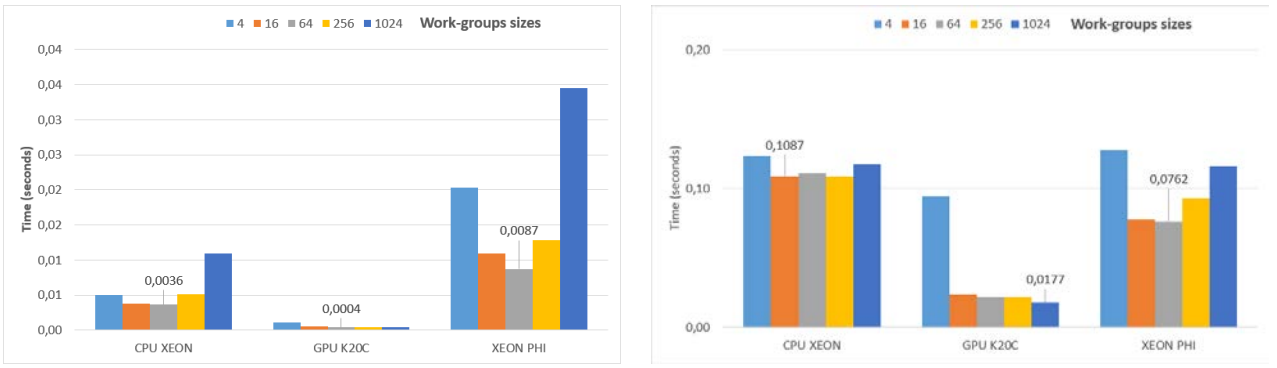
4.2 Performance Evaluation

In order to evaluate the performance portability study of the proposed parallel implementation, we are going to evaluate if a single code written in OpenCL allows us to achieve acceptable performance. Table 1 shows the times related with each stage, where *Initialization* includes the load image and allocates of memory, *Serial* corresponds to the execution time for the code not parallelized. *Brightest_pixel_spectra*, *Pixel_projection* and *Reduction_projection* denote the execution time for each kernel implemented. Finally, *ReadAndWrite memory* includes the transfer times from host to device and vice-versa and *Total Time* corresponds to the complete execution time excluding the *Initialization* time.

The performance results with respect to distribution work-load are shown in Fig.1, where bars display execution times for different work-groups sizes (4 to 1024). As can be seen, work-group size has a significant impact on performance in both GPU and Xeon-Phi device, meanwhile is negligible in multicore. On the other hand, the OpenCL implementation on the GPU platform outperforms the another counterparts being the best alternative using a single code written in OpenCL.

Table 1. Processing times (in seconds) and speedups achieved for the proposed OpenCL implementation in different platforms and tested with real data sets.

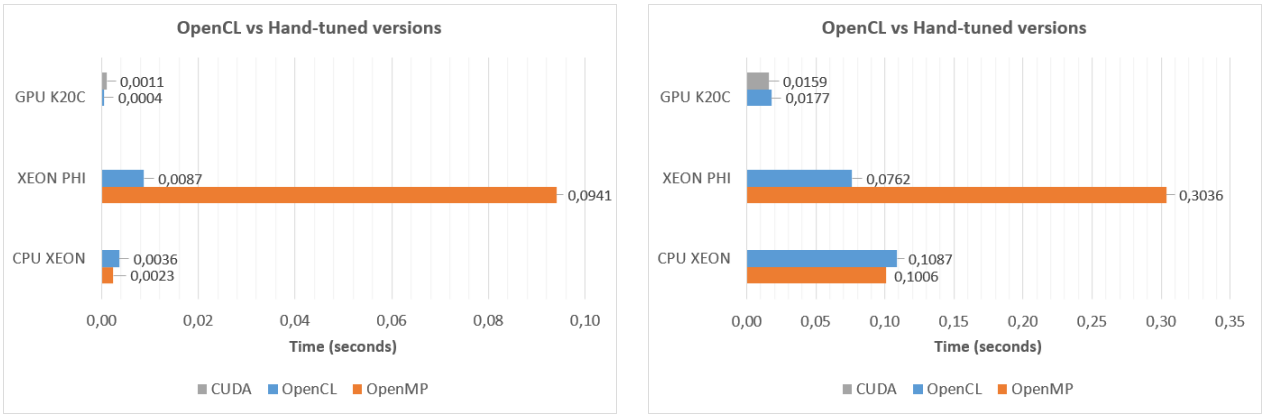
	Hydice				AVIRIS Cuprite			
	Serial CPU	CPU Xeon	GPU K20c	Xeon Phi	Serial CPU	CPU Xeon	GPU K20c	Xeon Phi
Initialization	0.0138	0.3523	0.2596	1.2544	0.0918	0.4550	0.3700	1.4178
Serial	≈ 0.0000 (0.25%)	≈ 0.0000 (0.63%)	≈ 0.0000 (4.74%)	≈ 0.0000 (0.27%)	0.0001 (0.00%)	0.0003 (0.26%)	0.0001 (0.75%)	0.0002 (0.28%)
Brightest_pixel_spectra	0.0028 (16.62%)	0.0004 (10.89%)	0.0001 (11.78%)	0.0027 (31.17%)	0.0614 (5.71%)	0.0051 (4.71%)	0.0008 (4.69%)	0.0051 (6.64%)
Pixel_projection	0.0134 (82.89%)	0.0023 (63.93%)	0.0002 (45.92%)	0.0032 (36.03%)	1.0113 (94.09%)	0.0957 (88.08%)	0.0146 (82.72%)	0.0563 (73.93%)
Reduction_projection	0.0000 (0.20%)	0.0008 (21.75%)	0.0001 (24.06%)	0.0018 (20.75%)	0.0018 (0.20%)	0.0049 (4.47%)	0.0006 (3.46%)	0.0095 (12.39%)
ReadAndWrite memory	–	0.0001 (2.79%)	0.0001 (13.54%)	0.0010 (11.78%)	–	0.0027 (2.47%)	0.0015 (8.37%)	0.0051 (6.73%)
Total Time	0.0162	0.0036	0.0004	0.0087	1.0746	0.1087	0.0177	0.0762
Speedup	–	4.47x	38.07x	1.86x	–	9.89x	60.87x	14.10x



(a) (b)

Figure 1. Execution times achieved on CPU Xeon, NVIDIA's GPU and Xeon Phi. (a) HYDICE scene and (b) AVIRIS Cuprite scene.

Finally, Fig.2 provides a comparison between our portable OpenCL code and the hand-tuned versions. The best case on the GPU, Xeon-Phi and Xeon architectures with CUDA and OpenMP are using blocks of 1024 threads, 32 threads and 16 threads, respectively for Hydice scene. On the other hand, we have blocks of 1024 threads, 100 threads and 16 threads, respectively for Cuprite scene. As can be seen, the best performance is obtained using the hand-tuned version for each platform. However, OpenCL can be a good choice if we need a fast implementation in a short time.



(a) (b)

Figure 2. Execution times achieved on CPU Xeon, NVIDIA's GPU and Xeon-Phi between the portable OpenCL code and the hand-tuned versions. (a) HYDICE scene and (b) AVIRIS Cuprite scene.

5. CONCLUSION AND FUTURE LINES

In this work, we give an evaluation of OpenCL as a parallel programming framework for hyperspectral image analysis compared with CUDA and OpenMP. The quantitative evaluation results indicate that the sustained

performance of every OpenCL program is lower than that of the equivalent CUDA or OpenCL program except with the Xeon-Phi where is necessary to exploit another parallelism mechanisms, for example, using vectorization. Therefore, although OpenCL allows us to access various compute devices in a unified manner, we have to further optimize the code for each of those devices. We are planning to explore the automatic performance tuning methodology based on profiling to enhance the performance portability of OpenCL applications, specially on Xeon-Phi and FPGAs devices where is necessary more optimization techniques.

6. ACKNOWLEDGEMENT

This work has been supported by the Spanish Ministry of Economy and Competitiveness (MINECO) through the research contract TIN 2012-32180 and the Formación Posdoctoral programme (FPDI-2013-16280).

REFERENCES

1. A. Plaza and C.-I. Chang, *High Performance Computing in Remote Sensing*, Taylor & Francis: Boca Raton, FL, 2007.
2. Y. Tarabalka, T. V. Haavardsholm, I. Kasen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing," *Journal of Real-Time Image Processing* **4**, pp. 1–14, 2009.
3. Y. Zhang, M. S. II, and A. A. Chien, "Improving performance portability in OpenCL programs," *Journal of Supercomputing* **7905**, pp. 136–150, 2013.
4. H. Ren and C.-I. Chang, "Automatic spectral target recognition in hyperspectral imagery," *IEEE Transactions on Aerospace and Electronic Systems* **39**(4), pp. 1232–1249, 2003.
5. S. Bernabe, S. Lopez, A. Plaza, and R. Sarmiento, "GPU Implementation of an Automatic Target Detection and Classification Algorithm for Hyperspectral Image Analysis," *IEEE Geoscience and Remote Sensing Letters* **10**(2), pp. 221–225, 2013.
6. S. Bernabe, S. Lopez, A. Plaza, R. Sarmiento, and P. G. Rodriguez, "FPGA design of an automatic target generation process for hyperspectral image analysis," *In Proceedings of the IEEE International Conference on Parallel and Distributed Systems* , pp. 1010–1015, 2011.
7. R. O. Green *et al.*, "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," *Remote Sensing of Environment* **65**(3), pp. 227–248, 1998.
8. J. C. Harsanyi and C.-I. Chang, "Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection," *IEEE Transactions on Geoscience and Remote Sensing* **32**(4), pp. 779–785, 1994.
9. S. Bernabe, S. Sanchez, A. Plaza, S. Lopez, J. A. Benediktsson, and R. Sarmiento, "Hyperspectral unmixing on GPUs and multi-core processors: a comparison," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* **6**(3), pp. 1386–1398, 2013.
10. C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*, Kluwer: New York, 2003.
11. C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.* **42**(3), pp. 608–619, 2004.