Portability Study of an OpenCL Algorithm for Automatic Target Detection in Hyperspectral Images

Sergio Bernabé, Carlos García, Francisco D. Igual, Guillermo Botella, Manuel Prieto-Matias, and Antonio Plaza, *Fellow, IEEE*

***Abstract*— In the last decades, the problem of target detection has received considerable attention in remote sensing applications. When this problem is tackled using hyperspectral images with hundreds of bands, the use of high-performance computing (HPC) is essential. One of the most popular algorithms in the hyperspectral image analysis community for this purpose is the automatic target detection and classification algorithm (ATDCA). Previous research has already investigated the mapping of ATDCA on HPC platforms such as multicore processors, graphics processing units (GPUs), and field-programmable gate arrays (FPGAs), showing impressive speedup factors (after careful fine-tuning) that allow for its exploitation in time-critical scenarios. However, the lack of standardization resulted in most implementations being too specific to a given architecture, eliminating (or at least making extremely difficult) code reusability across different platforms. In order to address this issue, we present a portability study of an implementation of ATDCA developed using the open computing language (OpenCL). We focus on cross-platform parameters such as performance, energy consumption, and code design complexity, as compared to previously developed (hand-tuned) implementations. Our portability study analyzes different strategies to expose data parallelism as well as enable the efficient exploitation of complex memory hierarchies in heterogeneous devices. We also conduct an assessment of energy consumption and discuss metrics to analyze the quality of our code. The conducted experiments—using synthetic and real hyperspectral data sets collected by the Hyperspectral Digital Imagery Collection Experiment (HYDICE) and NASA's Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS)—demonstrate, for the first time in the literature, that portability across different HPC platforms can be achieved for real-time target detection in hyperspectral missions.**

***Index Terms*— Automatic target detection and classification algorithm (ATDCA), code quality, energy consumption, high-performance computing (HPC), hyperspectral imaging, open computing language (OpenCL), portability.**

Manuscript received December 10, 2018; revised May 13, 2019; accepted June 18, 2019. Date of publication August 1, 2019; date of current version October 31, 2019. This work was supported in part by EU (FEDER) and in part by the Spanish MINECO under Grant TIN2015-65277-R, Grant TIN2015-63646-C5-5-R, and Grant RTI2018-093684-B-I00. *(Corresponding author: Sergio Bernabé.)*

S. Bernabé, C. García, F. D. Igual, G. Botella, and M. Prieto-Matias are with the Department of Computer Architecture and Automation, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: sebernab@ucm.es).

A. Plaza is with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, 10003 Cáceres, Spain.

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TGRS.2019.2927077

I. Introduction

HIGH-performance computing (HPC) has been used to map hyperspectral image analysis algorithms on many remote sensing applications [1], including environmental modeling, biological threat detection, monitoring of oil spills, target detection for military and defense/security purposes, wildfire tracking, and so on. This solution is needed because hyperspectral images are composed of hundreds or even thousands of spectral bands (at different wavelength channels), and there are important computational requirements to manage, process, and even store these high-dimensional data. One of the most popular imaging spectrometers currently in operation is the Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS), managed by NASA's Jet Propulsion Laboratory in California, which is able to record the visible and near-infrared spectrum (wavelength region from 0.4 to 2.5 μm) of reflected light in an area of 2 to 12 km wide and several kilometers long, using 224 spectral bands. The resulting multidimensional data cube typically comprises several gigabytes per flight.

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 emerging HPC computing platforms. Unfortunately, programming heterogeneous HPC systems is a laborious task that often involves a deep knowledge of the underlying architecture, as well as different programming languages. In fact, programmers are usually forced to concentrate on implementation details and learning new application interfaces (APIs) rather than on other important issues related to the application. Furthermore, the lack of standardization in the first generation products from International Business Machines Corporation (IBM) [Cell Broadband Engine (Cell BE) Architecture] 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. In turn, the open computing language (OpenCL) has become a free standard for parallel programming of heterogeneous systems.

The development of OpenCL was driven by the need to improve portability [3]–[5]. OpenCL is 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 sorts 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 using OpenCL is the shorter time to develop fast implementations. However, OpenCL makes no guarantee of performance portability. Code portability is important for application and library developers, but performance portability is what really matters to application users. Our focus in this paper is on analyzing and studying this emerging problem for the first time in the remote sensing literature, using as a benchmark the automatic target detection and classification algorithm (ATDCA) [6], 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 on HPC architectures, such as multi-core processors [7], GPUs [8], and FPGAs [9]. In these studies, different implementations of the algorithm have been considered, including a version using an orthogonal projection operator [orthogonal subspace projections (OSP)] and another version using the Gram–Schmidt (GS) method for orthogonalization, showing impressive speedup factors that allow the exploitation of the algorithm in time-critical scenarios. Based on these previous studies, this paper explores the portability of a novel OpenCL implementation across a range of processing devices, namely, multicore processors, GPUs, and accelerators. This approach clearly differs from previous works, focused on achieving the optimal performance on each platform. Here, we are more interested in the following issues: 1) to evaluate if a single code written in OpenCL allows us to achieve acceptable performance across all of the available platforms; 2) to assess the gap between our portable OpenCL code and the previously investigated hand-tuned versions; 3) to measure the energy consumption for each platform; and 4) to develop metrics able to evaluate the code quality. In addition, our study also includes the analysis of different tuning techniques that expose data parallelism using open multiprocessing (OpenMP) and compute unified device architecture (CUDA), to enable an efficient exploitation of the complex memory hierarchies found in these heterogeneous devices.

Our experimental results, conducted using synthetic and real hyperspectral data sets collected by the Hyperspectral Digital Imagery Collection Experiment (HYDICE) and NASA's AVIRIS sensor, demonstrate the importance of performance, power consumption, and code quality parameters. In our opinion, the analysis presented here is highly innovative and quite important in order to really calibrate the possibility of using heterogeneous HPC platforms for efficient hyperspectral image processing in real remote sensing missions.

The main contribution of this paper is the development of a new portable OpenCL implementation of the ATDCA algorithm for hyperspectral target detection, and its detailed cross-portability evaluation based on different parameters such as performance, power consumption, and code quality. To the best of authors' knowledge, this kind of study has not been previously conducted on a real application. This work extends previous results in [10], which were mainly focused on performance portability. Here, we extend the previous preliminary work along the following main lines.

1) Our new portable OpenCL version has been substantially improved. Among the new improvements, we provide optimal data distribution capability depending on the considered platform (SoA-based on pixel arrangements and AoS-based on spectral band arrangements), the use of padding to improve the performance with respect to the previous work and restrict to allow pointer disambiguation.
2) A previously available OpenMP hand-tuned version has also been improved considering a data distribution strategy based on the spectral bands arrangement, allowing for optimal memory transfers by means of memory alignment. This strategy also increases the efficiency of data loads and stores to and from the processor in architectures such as the Intel Xeon Phi platform. Moreover, this version guarantees cache coherence through data structures depending on the size of the cache memory.
3) Our performance study has been significantly extended, using a large synthetic image. In this study, we have enabled autovectorization flags, where the Intel compiler identifies and optimizes code portions, without requiring any special action by the programmer. This approach, together with the new data distribution strategy, improves the performance on most platforms studied.
4) Our power consumption study has been configured using a `pmlib` client/server infrastructure [11] by instrumenting the code using its API for energy consumption analysis of heterogeneous platforms.
5) Last but not least, a code quality study has been included to analyze our OpenCL portable code as compared to some previously available hand-tuned versions, using a newly developed metric.

The remainder of this paper is organized as follows. Section II describes our optimized ATDCA method. Section III describes the considered parallel implementations (CUDA, OpenMP, and OpenCL). Section IV presents an experimental evaluation of the considered implementations in terms of accuracy, parallel performance, power consumption, and code quality, using synthetic and real data sets, on heterogeneous HPC platforms. Finally, Section V concludes this paper with some remarks and hints at plausible future research lines.

II. Case Study: ATDCA-GS Method

The ATDCA is a popular algorithm in many remote sensing applications [2]. It was originally developed in [12] to find distinct targets using orthogonal subspace projections (OSP). This method has been modified in the last years [8], [13], [14]. In this work, we use an optimization of this algorithm (see [8]), 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 optimization is based on the GS method for orthogonalization.

In this work, we have focused on the ATDCA-GS algorithm because this algorithm can obtain real-time performance using

HPC platforms and even improving the accuracy compared to another ATDCA-based algorithms such as ATDCA-OSP [12]. In addition, ATDCA-GS is a simple algorithm but, at the same time, highly robust, with operations not very expensive in computational terms, i.e., the matrix inversion was removed and replaced by using the Gram–Smith method to obtain the orthogonal projections if we compare it with the ATDCA-OSP. The pseudocode of the ATDCA-GS algorithm is given in Algorithm 1.

Algorithm 1 Pseudocode of ATDCA-GS

1: **INPUTS:** $\mathbf{X} \in \mathbf{R}^n$ and t;
   % $\mathbf{X}$ denotes an n-dimensional hyperspectral image with n_s pixels and t denotes the number of targets to be detected
2: $\mathbf{M} = [\mathbf{x}_0 \mid 0 \mid, \ldots, \mid 0]$; % whose size is t
   % $\mathbf{x}_0$ is the pixel vector with maximum length in $\mathbf{X}$
3: $\mathbf{B} = [0 \mid 0 \mid, \ldots, \mid 0 \mid]$; % whose size is $t-1$
   % $\mathbf{B}$ is an auxiliary matrix for storing the orthogonal base generated by the GS process
4: **for** $i = 1$ to $t-1$ **do**
5: $\mathbf{B}[:, i] = \mathbf{M}[:, i]$;
   % the i-th column of $\mathbf{B}$ is initialized with the target computed in the last iteration (here, the operator ":" denotes "all elements")
6: $P_{\mathbf{M}}^{\perp} = [\mathbf{1}, \ldots, \mathbf{1}]$;
7: **for** $j = 2$ to i **do**
8: $proj_{\mathbf{B}[:,j-1]}(\mathbf{M}[:, i]) = \frac{\mathbf{M}[:,i]^T \mathbf{B}[:,j-1]}{\mathbf{B}[:,j-1]^T \mathbf{B}[:,j-1]} \mathbf{B}[:, j-1]$;
9: $\mathbf{B}[:, i] = \mathbf{B}[:, i] - proj_{\mathbf{B}[:,j-1]}(\mathbf{M}[:, i])$;
   % The i-th column of $\mathbf{B}$ is updated
10: **end for** j
   % The computation of $\mathbf{B}$ is finished for the current iteration of the main loop
11: **for** $k = 1$ to i **do**
12: $proj_{\mathbf{B}[:,k]}(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{B}[:,k]}{\mathbf{B}[:,k]^T \mathbf{B}[:,k]} \mathbf{B}[:, k]$;
13: $P_{\mathbf{M}}^{\perp} = P_{\mathbf{M}}^{\perp} - proj_{\mathbf{B}[:,k]}(\mathbf{w})$;
14: **end for** k
   % The computation of $P_{\mathbf{M}}^{\perp}$ is finished for the current iteration of the main loop
15: $\mathbf{v} = P_{\mathbf{M}}^{\perp} \mathbf{F}$;
   % $\mathbf{X}$ is projected onto the direction indicated by $P_{\mathbf{M}}^{\perp}$
16: $i = \text{argmax}_{\{1,\ldots,n_s\}} \mathbf{v}[:, i]$;
   % The maximum projection value is found
17: $\mathbf{x}_i \equiv \mathbf{M}[:, i+1] = \mathbf{X}[:, i]$;
   % The target matrix is updated
18: **end for**
19: **OUTPUT:** $\mathbf{M} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}]$;
   % $\mathbf{M}$ denotes the matrix with all the spectral signatures for each target

III. Parallel Implementations

In this section, we describe our parallel versions of the ATDCA-GS algorithm for a diverse set of heterogeneous platforms. The OpenMP and CUDA implementations are first presented. Section III-B describes our portable OpenCL implementation that represents the main focus of this study.

A. OpenMP Implementation

By observing the program flow in Algorithm 1 [10], it is possible to identify the potential bottlenecks in the ATDCA-GS algorithm. The most important one is the projection of the orthogonal vector onto each pixel spectrum on the image, where a reduction process is included. We have identified such bottleneck by profiling an optimized (serial) ATDCA-GS implementation.

First, the hyperspectral image $\mathbf{X}$ is mapped onto RAM memory. In this case, the best data distribution is based on the spectral bands arrangement, where each spectral band vector with size n_s is distributed (by columns) in $\mathbf{X}$, allowing optimal memory transfers to create data objects with starting addresses that are modulo 64 bytes. As we will see later, this scheme does not offer the best performance rates on GPU devices.

After that, it is needed to calculate the brightest pixel spectra $\mathbf{x}_0$ in $\mathbf{X}$. This pixel is the initial target signature whose pixel vector contains the pixel with maximum projection value in the original n-dimensional hyperspectral image. In this step, the algorithm computes a dot product between each pixel vector and its own transposed version. The parallel algorithm has been rewritten in order to avoid the use of locking routines (*omp_set_lock*) and (*omp_unset_lock*), as shown in lines 11–14 of Algorithm 2. Note that we have also solved the false sharing overhead by means of padding technique.

Algorithm 2 OpenMP Brightest_Pixel_Spectra Calculation

1: global h_image ← Initial $\mathbf{X}$ vector [r]
   max_locals[MAX_THREADS*CACHE_LINE] ← 0,
   pos_abs_locals[MAX_THREADS*CACHE_LINE] ← 0
   % MAX_THREADS denotes the maximum number to execute the code
   % CACHE_LINE denotes the maximum size for each cache line to guarantee cache coherence
2: registers max_local ← 0, value ← 0, value_out ← 0
3: #*pragma* omp parallel for private(value, value_out, j)
4: **for** iter=0 to r **do**
5: th ← omp_get_thread_num()
6: value ← 0, value_out ← 0
7: **for** $j = 0$ to n_b **do**
8: value ← h_image[j + (iter * n_bPADDING)]
9: value_out ← value * value
10: **end for**
11: **if** value_out > max_locals[th*CACHE_LINE] **then**
12: max_locals[th*CACHE_LINE] ← value_out
13: pos_abs_locals[th*CACHE_LINE] ← iter
14: **end if**
15: **end for**
16: **for** $iter = 0$ to MAX_THREADS **do**
17: **if** max_locals[iter*CACHE_LINE] > max_local **then**
18: max_local ← max_locals[iter*CACHE_LINE]
19: pos_abs ← pos_abs_locals[iter*CACHE_LINE]
20: **end if**
21: **end for**

Once the brightest pixel in $\mathbf{X}$ has been identified, its spectral signature is allocated as the first column in matrix $\mathbf{M}$. In order

Algorithm 3 OpenMP Pixel_Projection Calculation

```
1: global h_image ← Initial X vector [r]
2: global h_f ← The most orthogonal vector
3: global max_locals[MAX_THREADS*
   CACHE_LINE_FLOAT] ← 0,
   pos_abs_locals[MAX_THREADS*CACHE_LINE_INT]
   ← 0
4: registers max_local ← 0, value ← 0, value_out ← 0
5: #pragma omp parallel for private(value, value_out, j)
   schedule(guided)
6: for iter=0 to r do
7:   th ← omp_get_thread_num()
8:   value ← 0, value_out ← 0
9:   for j=0 to n_b do
10:    value ← value + h_image[j + (iter * n_bPADDING)]
       * h_f[j]
11:  end for
12:  value_out ← value * value
13:  if value_out > max_locals[th*CACHE_LINE_FLOAT]
     then
14:    max_locals[th*CACHE_LINE_FLOAT] ← value_out
15:    pos_abs_locals[th*CACHE_LINE_INT] ← iter
16:  end if
17: end for
18: for iter = 0 to MAX_THREADS do
19:  if max_locals[iter*CACHE_LINE_FLOAT] >
     max_local then
20:    max_local ← max_locals[iter*CACHE_LINE_FLOAT]
21:    pos_abs ← pos_abs_locals[iter*CACHE_LINE_INT]
22:  end if
23: end for
```

to calculate the orthogonal vectors through the GS method, serial version is used. It permits to operate on small data structures and obtain the results very quickly. Then, a parallel function is created to project the orthogonal vector onto each pixel of the image and obtain the maximum of all projected pixels. The function is outlined in Algorithm 3, where we have used the same strategy as in Algorithm 2. The considered data distribution mechanism and the strategy adopted to solve the false sharing problem improve significantly the performance with respect to the previous research [10], since now we ensure cache coherence through data structures depending on the size of the cache memory, increasing the efficiency of our application.

Algorithm 1 now extends the target matrix as $\mathbf{M} = [\mathbf{x}_0\mathbf{x}_1]$ 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 $\mathbf{M} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}]$.

B. CUDA Implementation

Our parallel implementation of ATDCA-GS in CUDA is based on the strategy described in [10], but using the version 9.0 of CUDA. Three different kernels have been developed to calculate the brightest pixel (see Algorithm 4), pixel projection (see Algorithm 5) and a reduction process to obtain the maximum projection (see Algorithm 6). The main novelty of this implementation respect to [10] is the efficient exploitation of memory adding memory padding and two data distributions (see Lines 7 and 19 on Algorithms 4 and 5, respectively), where the best distribution is based on the number of pixels arrangement, in which each pixel vector is distributed by columns in $\mathbf{X}$.

Algorithm 4 CUDA Brightest_Pixel_Spectra Kernel

```
1: global d_image ← Initial X vector [r]
2: global d_bright ← The bright value for each pixel spectral
   x_i in X
3: registers bright ← 0, value ← 0
4: id ← blockDim.x * blockIdx.x + threadIdx.x
   % 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 * n_bPADDING)]
8:     bright ← bright + value * value
9:   end for
10:  d_bright[id] ← bright
11: end if
```

Algorithm 5 CUDA Pixel_Projection Kernel

```
1: global d_image ← Initial X vector [r]
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: shared s_df[n_b] ← Initial d_f structure with the most
   orthogonal vector
6: idx ← blocDim.x * blockIdx.x + threadIdx.x
7: if id < r then
8:   if threadIdx.x < n_b then
9:     for i = threadIdx.x to n_b do
10:      s_df[i] ← d_f[i]
11:    end for
12:  else
13:    if threadIdx.x < n_b then
14:      s_df[threadIdx.x] ← d_f[threadIdx.x]
15:    end if
16:  end if
17:  __syncthreads()
     % In this synchronize, all threads must wait the execution
     of all threads in a block to complete the copy of d_f
18:  for i = 0 to n_b do
19:    value ← d_image[idx + (i * n_bPADDING)]
20:    sum ← sum + value * s_df[i]
21:  end for
22:  d_projection[idx] ← sum * sum
23: end if
```

C. OpenCL Implementation

Our portable implementation in OpenCL has several differences with the implementation in [10]. Before analyzing them, we briefly describe the OpenCL framework.

Algorithm 6 CUDA Reduction_Projection Kernel

```
global d_bright ← The bright value for each pixel spectral in X
global d_projection ← The projection value for each pixel spectral x_i in F
global d_index ← The index for each projection value
local s_p ← Initial structure to store all the projections
local s_i ← Initial structure to store all the index for each projection
tid ← threadIdx.x
i ← blockIdx.x * (blockDim.x * 2) + tid
if (i+blockDim.x) ≥ r then
  s_p[tid] ← d_bright[i]
  s_i[tid] ← i
else
  if d_bright[i] > d_bright[i + blockDim.x] then
    s_p[tid] ← d_bright[i]
    s_i[tid] ← i
  else
    s_p[tid] ← d_bright[i + blockDim.x]
    s_i[tid] ← i + blockDim.x
  end if
end if
__syncthreads()
% In this synchronization, all threads must wait for the execution of all threads in a block to complete the copy in the local memory of s_p and s_i
for s=blockDim.x / 2 to s>0 do
  if tid < s then
    if s_p[tid] ≤ s_p[tid + s] then
      s_p[tid] ← s_p[tid + s]
      s_i[tid] ← s_i[tid + s]
    end if
  end if
  __syncthreads()
end for
d_projection[blockIdx.x] ← s_p[0]
d_index[blockIdx.x] ← s_i[0]
__syncthreads()
```

1) OpenCL Framework: The OpenCL specification was developed to allow freedom in terms of implementing applications that may be run on many add-ons that are vendor-specific, cross vendor, and Khronos.[1] In the OpenCL paradigm, "host program" is in charge of I/O operations, data initialization, and "device" control. It launches the "kernel" codes and synchronizes them. Among the main benefits, there are the wide range of hardware devices such as CPUs, GPUs, and FPGAs that could be used easily with a moderate coding effort.

An OpenCL kernel allows to express parallelism by means of the execution of several work-items. A group of work-items forms a work-group that runs on a single compute unit. The work-items execute the same kernel (with a unique id), share a fast memory denoted as local memory, and can be synchronized with barriers. The maximum dimension of each work-group depends on the specifications of the device in use (see Section IV).

2) Implementation Details: In our implementation, the use of padding, restrict, and two data distributions is tested. Although the kernels are written in a similar way as in CUDA implementations, the data arrangement in memory is different. The best performance rates observed correspond to the data distribution based on the spectral bands arrangement. We would like to note that this scheme offers better performance on Xeon platforms because it favors the use of native OpenCL vectorization. However, this memory access pattern is not optimal on GPUs since it is performed in a noncoalesced way. In addition, we have focuses on efficient memory exploitation since the algorithm is memory-bound. According to the number of memory access, its arithmetic intensity is about 1. Memory optimization has been performed (especially on Algorithm 8) by means of data reuse in the local memory and applying blocking techniques.

The differences with respect to the implementation in [10] are highlighted in Algorithms 7 (brightest pixel) and 8 (pixel projection). The reduction kernel is not shown because we have used the same scheme as in CUDA described in Algorithm 6. It should be noted that it was not possible to perform any task parallelism because the algorithm exhibits data dependencies the stages mentioned.

Algorithm 7 OpenCL Brightest_Pixel_Spectra Kernel

```
global restrict d_image ← Initial X vector [r]
global restrict d_bright ← The bright value for each pixel spectral x_i in X
% restrict avoids that the pointers do not point to overlapping locations.
registers bright ← 0, value ← 0
id ← get_group_id(0) * get_local_size(0) + get_local_id(0)
% n_b denotes the number of spectral bands
if id < r then
  for k = 0 to n_b do
    value ← d_image[k + (id * n_bPADDING)]
    bright ← bright + value * value
  end for
  d_bright[id] ← bright
end if
```

IV. Experimental Results

This section is organized as follows. Section IV-A provides the work environment used in our experiments. Section IV-B describes the hyperspectral data sets used in this study. Section IV-C evaluates the target detection accuracy of the considered implementations. Finally, Sections IV-D–IV-F, respectively, discuss the performance, power consumption, and code quality evaluations.

A. Considered Work Environment

The experiments have been carried out in three different types of systems. The first two are HPC heterogeneous systems with the same host and different accelerator/coprocessors,

[1] https://www.khronos.org/opencl

Algorithm 8 OpenCL Pixel_Projection Kernel

```
global restrict d_image ← Initial X vector [r]
global restrict d_projection ← The projection value for each pixel spectral x_i in X
global restrict d_f ← The most orthogonal vector
registers sum ← 0, value ← 0
local s_df[n_b] ← Initial d_f structure with the most orthogonal vector
id ← get_global_id(0)
if id < r then
  if get_local_size(0) < n_b then
    for i = get_local_id(0) to n_b do
      s_df[i] ← d_f[i]
    end for
  else
    if get_local_id(0) < n_b then
      s_df[get_local_id(0)] ← d_f[get_local_id(0)]
    end if
  end if
  barrier(CLK_LOCAL_MEM_FENCE)
  % Wait until the copy in local memory of d_f is completed
  for i = 0 to n_b do
    value ← d_image[i + (id * n_bPADDING)]
    sum ← sum + value * s_df[i]
  end for
  d_projection[id] ← sum * sum
end if
```

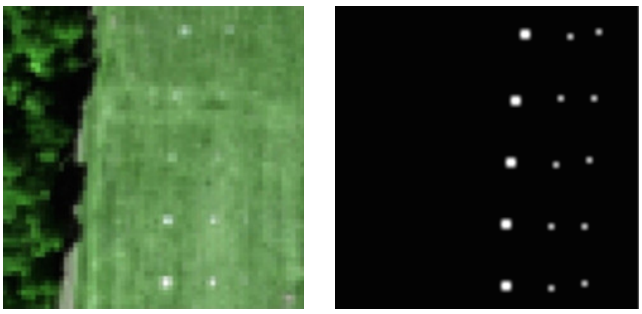
while the last one is a low-power environment. Their main features are described in the following items.

1) The host of the two HPC heterogeneous systems consists of two Intel Xeon E5–2695 processors with 14 cores each at 2.30 GHz and 64 GB of DDR3 RAM memory. Two types of accelerators have been considered as follows.
   a) An NVIDIA GeForce GTX 1080 GPU with 2560 cores operating at 1772 MHz and 8 GB of dedicated GDDR5X memory.
   b) A Xeon-Phi 31S1P coprocessor with 57 cores supporting the execution of four hardware threads (228 hardware threads in total) operating at 1100 MHz and 8-GB RAM memory.
2) An additional low-power platform is used in our experiments, with the following main features.
   a) ODROID-XU3 platform equipped with Samsung Exynos5422 Cortex-A15 2.0-GHz quad-core and Cortex-A7 quad-core CPUs and Mali-T628 MP6 GPU and 2-GB LPDDR3 RAM memory operating at 933 MHz.

B. Real and Synthetic Data Sets

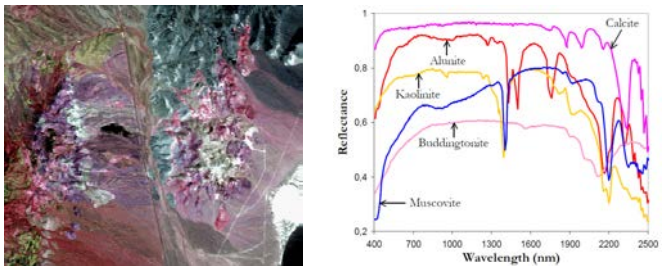
We have considered three hyperspectral data sets in our experiments.

1) The first one is a scene collected by the HYDICE sensor [see Fig. 1(a)], which represents a subset of the well-known forest radiance data set, consisting of 64×64 pixels with 15 panels and 169 spectral bands, for a total size of 5.28 MB. Moreover, a ground-truth map is available for the scene, indicating the spatial location of the panels [see Fig. 1(b)]. This image was acquired with 210 spectral bands and a spectral coverage from 0.4 to 2.5 μm. Bands 1–3, 101–112, 137–153 and 202–210 were removed prior to the analysis due to water absorption and low signal-to-noise ratio (SNR) in those bands. The spatial resolution of the scene is 1.56 m. The scene is extensively described in [15].
2) The second hyperspectral image scene is the well-known AVIRIS Cuprite scene [see Fig. 2(a)], collected by the NASA/JPL AVIRIS spectrometer in the summer of 1997. It is available online in reflectance units after atmospheric correction and comprises a relatively large area (350 lines by 350 samples and 20-m pixels) and 224 spectral bands between 0.4 to 2.5 μm, with a total size of around 46 MB. Bands 1–3, 105–115, and 150–170 were removed prior to the analysis due to water absorption and low SNR in these bands. The site is well understood mineralogically and has several exposed mineral of interest including alunite, buddingtonite, calcite, kaolinite, and muscovite. The ATDCA-GS algorithm has been assessed in this work using the reference ground signatures of the above-mentioned minerals, displayed in Fig. 2(b), which are available from the United States Geological Survey (USGS) library.[2]



(a) (b)

Fig. 1. (a) False color representation of the HYDICE hyperspectral scene. (b) Associated ground-truth information.



(a) (b)

Fig. 2. (a) False color composition of an AVIRIS hyperspectral image collected over the Cuprite mining district in Nevada. (b) USGS mineral spectral signatures used for validation purposes.

[2]United States Geological Survey (USGS) library: http://speclab.cr.usgs.gov/spectral-lib.html

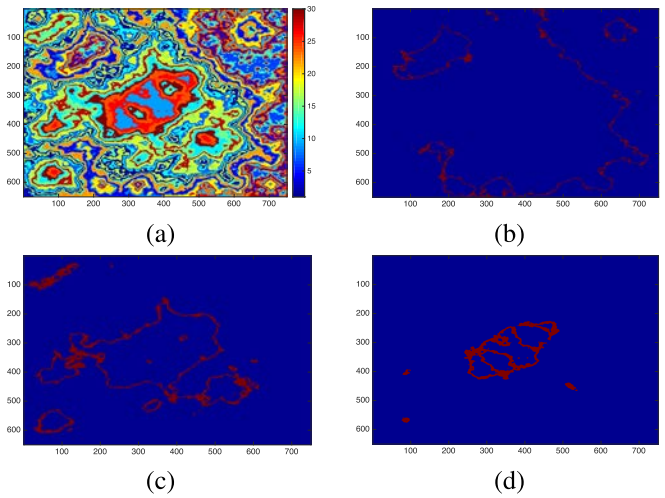


Fig. 3. (a) Color-scale composition and three examples of ground-truth abundance maps of endmembers (pure spectral signatures) in the synthetic hyperspectral data. (b) Endmember #3. (c) Endmember #15. (d) Endmember #26.

3) We also considered a bigger synthetic data set in order to evaluate the scalability of our implementations. The data have been constructed using a set of 30 signatures from the USGS library and the procedure described in [16] to simulate natural spatial patterns. The resulting synthetic image is composed by a total of 750 × 650 pixels, resulting in a size of around 437 MB. Fig. 3 displays a color-scale composition and three examples of ground-truth abundance maps (built from pure spectral signatures or endmembers) for this simulated image.

C. Accuracy Evaluation

It is important to emphasize that our parallel implementations of the ATDCA-GS provide exactly the same numerical results as their respective sequential implementations. In order to analyze the accuracy of the general ATDCA-GS algorithm, the well-known spectral angle distance (SAD) [15] is adopted in this work. For this purpose, we have chosen the AVIRIS Cuprite scene, which is highly used for evaluation purposes due to the availability of reference spectral signatures [see Fig. 2(b)]. The number of targets to be extracted was estimated as $t = 19$ after the consensus between two well-known techniques for this purpose: the virtual dimensionality (VD) [17] and the hyperspectral signal identification with minimum error (HySime) algorithm [18]. The SAD was calculated between the extracted targets and the ground-truth USGS spectral signatures. The range of values for the spectral angle is in the range [0°, 90°] (the lower the SAD value, the higher the spectral similarity). As shown in Table I, the endmembers extracted by the ATDCA-GS exhibit low SAD scores and are spectrally very similar to the USGS reference signatures, despite the potential variations (due to possible interference still remaining after the atmospheric correction process) between the ground signatures and the airborne data. These results indicate that good detection performance for all considered targets is achieved by ATDCA-GS.

TABLE I

SPECTRAL ANGLE VALUES (IN DEGREES) BETWEEN THE TARGET PIXELS EXTRACTED BY THE ATDCA-GS ALGORITHM AND THE REFERENCE USGS MINERAL SIGNATURES AVAILABLE FOR THE AVIRIS CUPRITE SCENE

Alunite	Buddingtonite	Calcite	Kaolinite	Muscovite	Average
5.448°	4.08°	5.87°	11.14°	5.68°	6.45°

D. Performance Evaluation

In this section, we conduct an evaluation of the computational performance of our portable OpenCL implementation in several heterogeneous platforms. Our developed codes have been compiled using the Intel ICC compiler v18.0 on the Intel Xeon host and GNU-GCC compiler v4.9 on the Odroid-XU3, with the enabled flags *-O3 -restrict* that activate single instruction multiple data (SIMD) exploitation.

Tables II–IV show the obtained times after processing the simulated and real data sets on the considered platforms, depending on the work-group size and two considered data distributions: SoA (structure of arrays) and AoS (array of structures). For the sake of clarity, the best times achieved in each configuration are highlighted in bold. The speedups are calculated using as baseline the best time obtained in the serial implementation on a single Xeon CPU. In this work, the storage resources are not a bottleneck and, as a result, we have focused on the use of different devices applied to different image sizes. As it can be observed, the SoA distribution is the most effective strategy on the GPU, XeonPhi, and Odroid platforms, where the optimal work-group size is close to the number of physical cores present on each device. In contrast, the AoS distribution behaves as an efficient alternative on general-purpose processors based on Xeon. Furthermore, it is also remarkable the scalability of our parallel implementations, as the speedups increase with the image size, achieving up to 11× speedup on the GPU device due to the efficient use of different computing capacities for each device (except for the Odroid platform, where the largest image does not fit in RAM memory).

In order to highlight the use of using OpenCL and the involved overheads, Figs. 4–6 show a breakdown of the measured execution times for each hyperspectral data set. The tag RAM->Device represents the overhead of transferring the image between the main memory of the host and the device. Serial code includes the code executed on sequential way: input/output (IO) operations and kernel launch overheads. Read_Write accounts the data transfer overheads between the host and device memory. The rest of the subbars show the execution times of each stage of ATDCA. It is important to note that the bar denoted as *Another P.P.L.* means the total time achieved by means of a native hand-tuned coding version in each device (OpenMP or CUDA).

With regards to Fig. 4, the most successful threads configuration considered for OpenMP and CUDA is 8 on Xeon CPU, 1024 on GPU, 4 on Xeon Phi, and 4 on the Odroid. Since this scene can be regarded as a small one, our OpenCL version achieves low-performance ratios with respect to the hand-tuned counterpart. This is expected,

TABLE II

PROCESSING TIMES (IN SECONDS) OBTAINED BY THE PROPOSED OPENCL IMPLEMENTATION OF ALGORITHM 1 IN DIFFERENT PLATFORMS, TESTED WITH THE HYDICE DATA SET

	HYDICE				
	Work-Group$_{size=4}$	**Work-Group$_{size=16}$**	**Work-Group$_{size=64}$**	**Work-Group$_{size=256}$**	**Work-Group$_{size=1024}$**
Serial_SoA	0.0223	0.0223	0.0223	0.0223	0.0223
Serial_AoS	0.0017	0.0017	0.0017	0.0017	0.0017
Xeon_SoA	0.0092	**0.0077**	0.0078	0.0099	0.0152
Xeon_AoS	0.0148	0.0111	0.0113	0.0113	0.0122
Speedup_Xeon	–	0.22x	–	–	–
GPU_SoA	0.0067	0.0054	0.0059	0.0058	**0.0033**
GPU_AoS	0.0047	0.0036	0.0037	0.0039	0.0037
Speedup_GPU	–	–	–	–	0.52x
XeonPhi_SoA	0.0297	0.0197	0.0179	**0.0170**	0.0209
XeonPhi_AoS	0.0224	0.0204	0.0179	0.0175	0.0292
Speedup_XeonPhi	–	–	–	0.10x	–
Odroid_SoA	0.0305	0.0305	0.0303	**0.0300**	–
Odroid_AoS	0.0343	0.0349	0.0349	0.0350	
Speedup_Odroid	–	–	–	0.06x	–

TABLE III

PROCESSING TIMES (IN SECONDS) OBTAINED BY THE PROPOSED OPENCL IMPLEMENTATION OF ALGORITHM 1 IN DIFFERENT PLATFORMS, TESTED WITH THE AVIRIS CUPRITE DATA SET

	AVIRIS Cuprite				
	Work-Group$_{size=4}$	**Work-Group$_{size=16}$**	**Work-Group$_{size=64}$**	**Work-Group$_{size=256}$**	**Work-Group$_{size=1024}$**
Serial_SoA	2.0301	2.0301	2.0301	2.0301	2.0301
Serial_AoS	0.2665	0.2665	0.2665	0.2665	0.2665
Xeon_SoA	0.0692	0.0837	0.0788	0.0820	0.0829
Xeon_AoS	**0.0592**	0.0594	0.0661	0.0773	0.0832
Speedup_Xeon	4.50x	–	–	–	–
GPU_SoA	0.0511	0.0400	0.0352	0.0340	**0.0339**
GPU_AoS	0.0487	0.0508	0.1245	0.1401	0.1101
Speedup_GPU	–	–	–	–	7.86x
XeonPhi_SoA	0.1570	0.0767	0.0759	**0.0738**	0.0764
XeonPhi_AoS	0.1212	0.0944	0.0998	0.1062	0.1175
Speedup_XeonPhi	–	–	–	3.61x	–
Odroid_SoA	1.0668	1.0653	1.0495	**0.9642**	–
Odroid_AoS	1.9586	1.9702	1.9825	1.4608	–
Speedup_Odroid	–	–	–	0.28x	–

TABLE IV

PROCESSING TIMES (IN SECONDS) OBTAINED BY THE PROPOSED OPENCL IMPLEMENTATION OF ALGORITHM 1 IN DIFFERENT PLATFORMS, TESTED WITH THE SYNTHETIC DATA SET

	Synthetic				
	Work-Group$_{size=4}$	**Work-Group$_{size=16}$**	**Work-Group$_{size=64}$**	**Work-Group$_{size=256}$**	**Work-Group$_{size=1024}$**
Serial_SoA	16.2014	16.2014	16.2014	16.2014	16.2014
Serial_AoS	1.7274	1.7274	1.7274	1.7274	1.7274
Xeon_SoA	0.3912	0.4234	0.4157	0.4196	0.4431
Xeon_AoS	**0.3056**	0.3872	0.3848	0.3869	0.4030
Speedup_Xeon	5.65x	–	–	–	–
GPU_SoA	0.2792	0.1942	0.1642	0.1611	**0.1536**
GPU_AoS	0.2607	0.2892	0.8276	0.8837	1.1769
Speedup_GPU	–	–	–	–	11.25x
XeonPhi_SoA	2.3167	0.5882	**0.5855**	0.5947	0.6442
XeonPhi_AoS	0.8736	0.7195	0.7102	0.7185	0.8065
Speedup_XeonPhi	–	–	2.95x	–	–
Odroid_SoA	NA	NA	NA	NA	NA
Odroid_AoS	NA	NA	NA	NA	NA
Speedup_Odroid	–	–	–	–	–

due to the small degree of fine-grained parallelism available, which does not compensate the kernel launch overheads. In this particular case, a hand-tuned version is much faster.

In Fig. 5, the most successful threads configurations for OpenMP and CUDA are, respectively, 24 on Xeon CPU, 1024 on the GPU, 40 on Xeon Phi, and 12 on the Odroid. This image is considered as a medium-sized one. In this scenario,

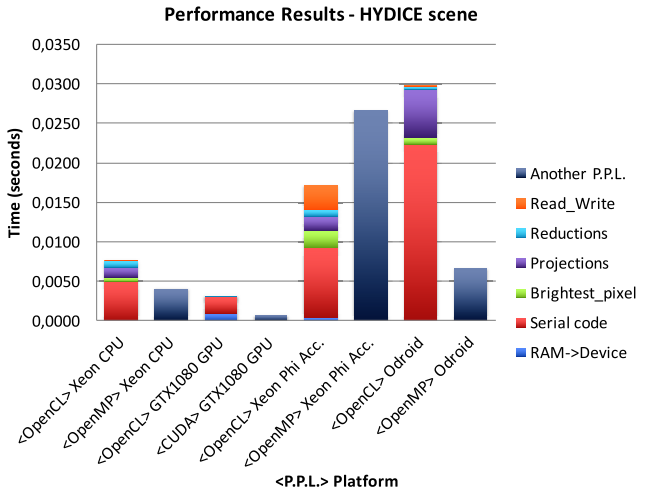


Fig. 4. Breakdown of the execution time considering all the parallel programming languages and platforms available (HYDICE data set).

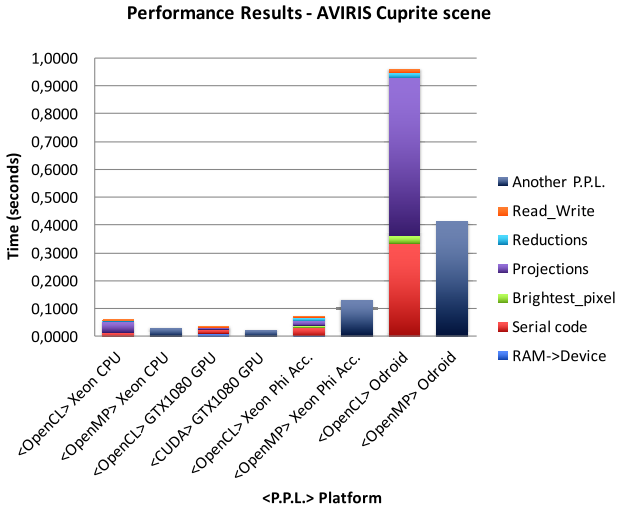


Fig. 5. Breakdown of the execution time considering all the parallel programming languages and platforms available (AVIRIS Cuprite data set).

our portable OpenCL code achieves performance ratios that are close to those exhibited by the hand-tuned versions in all considered cases, except on the Xeon Phi platform.

Finally, in Fig. 6, the most successful threads configurations are 28 on the Xeon CPU, 1024 on the GPU, and 57 on the Xeon Phi. Results in Odroid are not available since the scene cannot be allocated in the available memory of this device. Due to the fact that the size of this image is huge, the hand-tuned versions exhibit the best performance ratios. As in the previous analysis, our portable OpenCL version is able to attain similar performance than the one achieved by the hand-tuned counterparts on each platform. Thus, we conclude that the OpenCL implementation not only favors code reusability on different devices but can also achieve similar performance when compared to the hand-tuned versions on the considered devices.

Last but not least, we emphasize that our proposed portable version meets real-time requirements, i.e., it can process the considered scenes in less than 0.066 s (HYDICE), 1.986 s (AVIRIS), and 7.903 s (synthetic), taking as reference the cross-track line scan time in HYDICE and AVIRIS, which

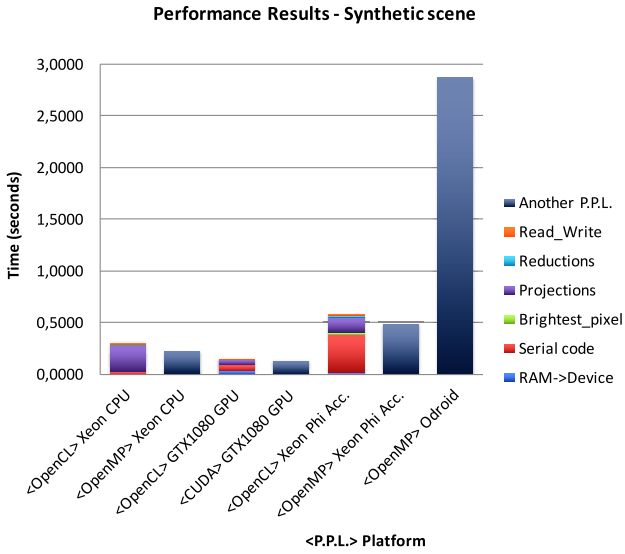


Fig. 6. Breakdown of the execution time considering all the parallel programming languages and platforms available (synthetic data set).

are push-broom instruments able to collect at least 512 full pixel vectors in 8.3 ms.

E. Energy Consumption Evaluation

In this section, we perform a detailed empirical study of the energy efficiency of our implementations in each considered architecture. We leverage the `pmlib` [11] client/server infrastructure as a common mechanism to measure energy consumption by instrumenting the code using its API. In all cases, the measured code in terms of energy matches that temporized in Section IV-D targeting performance.

In our case, the `pmlib` server has been adapted to gather power consumption from a specific mechanism depending on the target architecture given as follows.

1) *Intel Xeon CPU*: For this architecture, we use measurements offered by the Intel Running Average Power Limit (RAPL) infrastructure.
2) *Intel Xeon Phi*: We use the MicMgmt API offered by Intel.
3) *Nvidia GPU*: We use the Nvidia NVML library.
4) *Odroid XU3*: We gather instantaneous power consumption ratios from the internal sensors (Texas Instruments INA231) equipped in the board.

In all cases, we have measured instantaneous power consumption at the maximum rate allowed by the underlying mechanism, gathering an average value at the end of the execution.

Table V reports the obtained results on the target platforms. These results include total consumed energy (in terms of joules), average power dissipation (in terms of watts), and energy efficiency (in terms of Mpixels/watt). Comparing architectures, it is clear that the ODROID board is the most energy-efficient one for this family of implementations. Specifically, the use of OpenMP to leverage the CPU cores yields the most efficient implementations, followed by the GPU (OpenCL) implementation. Note, however, that this energy efficiency

TABLE V
COMPARATIVE STUDY OF ENERGY CONSUMPTION AND PERFORMANCE BETWEEN DIFFERENT PLATFORMS AND THE BEST SETTING FOR EACH PARALLEL PROGRAMMING LANGUAGE

Platforms	P.P.L.	Data Set	Time	J	Watt	Mpixels/Watt
Xeon	OpenCL	Hydice	0.0077	1.50	194.44	0.0026
		Cuprite	0.0592	11.58	195.53	0.0101
		Synthetic	0.3056	61.08	199.88	0.0076
	OpenMP	Hydice	0.0041	0.52	126.22	0.0075
		Cuprite	0.0321	5.98	186.37	0.0195
		Synthetic	0.2267	43.72	192.86	0.0106
GPU	OpenCL	Hydice	0.0033	0.40	120.99	0.0098
		Cuprite	0.0339	5.38	158.69	0.0217
		Synthetic	0.1536	25.28	164.59	0.0184
	CUDA	Hydice	0.0008	0.11	143.10	0.0341
		Cuprite	0.0249	4.27	171.30	0.0274
		Synthetic	0.1388	23.96	172.60	0.0194
Xeon_Phi	OpenCL	Hydice	0.0170	2.04	120.01	0.0019
		Cuprite	0.0738	9.72	131.74	0.0120
		Synthetic	0.5855	78.08	133.36	0.0060
	OpenMP	Hydice	0.0267	2.78	104.25	0.0014
		Cuprite	0.1331	15.53	116.65	0.0075
		Synthetic	0.4824	58.32	120.90	0.0080
Odroid	OpenCL	Hydice	0.0300	0.07	2.17	0.0600
		Cuprite	0.9642	3.03	3.14	0.0386
		Synthetic	–	–	–	–
	OpenMP	Hydice	0.0068	0.03	4.76	0.1207
		Cuprite	0.4166	2.12	5.08	0.0552
		Synthetic	2.8689	14.63	5.10	0.0318

comes at the expense of reducing performance by orders of magnitude, which is not always a feasible tradeoff. Also, observe how the ODROID platform, being a low-power one, yields the best power efficiency for small data sets, as opposed to the rest of the considered architectures. Specifically, among the remaining architectures (CPU, GPU, and Xeon Phi), GPUs yield the best energy efficiency results. For power-constrained scenarios, the reduced average power dissipated by the Xeon Phi can be of great appeal, despite its poor energy efficiency.

F. Code Quality Evaluation

This section addresses an attempt to measure the code development effort of the three parallel programming paradigms used in this paper: OpenCL, CUDA, and OpenMP. Thus, we can characterize the complexity in terms of the cost prior to subsequent coding. This is of high interest in terms of acquiring knowledge about the complexity achieved in the aforementioned parallel implementations in order to take future decisions.

We will use Maurice Halstead's [19]–[21] product hypothesis' effect derived from information theory and psychology. The metrics are based on the following four measurable properties of the code.

1) n_1: The number of distinct operators.
2) n_2: The number of distinct operands.
3) N_1: Operators count.
4) N_2: Operands count.

Using these input values, the following measures can be defined.

1) *Program Length (L):* $L = N_1 + N_2$.
2) *Program Vocabulary (V):* $V = n_1 + n_2$.
3) *Volume (VL):* $VL = L \times log_2(V)$.
4) *Difficulty (D):* $D = (n_1/n_2) \times (N_2/2)$.
5) *Effort (E):* $E = D \times V$.

TABLE VI
CODE QUALITY EVALUATION

Halstead metrics	OpenMP	OpenCL	CUDA
Operators count (N_1)	4,076.00	6,257.00	4,361.00
Distinct operators (n_1)	48.00	64.00	49.00
Operands count (N_2)	2,177.00	3,416.00	2,230.00
Distinct operands (n_2)	257.00	429.00	335.00
Program length (L)	6,253.00	9,673.00	6,591.00
Program vocabulary (V)	305.00	493.00	384.00
Volume (VL)	51,603.92	86,529.28	56,583.49
Difficulty (D)	203.00	254.00	159.00
Effort (E)	10,475,595.14	21,978,436.67	8,996,774.57

Halstead's length (L) simply adds up the number of unique operators and operands used in the code. A small number of statements with a high volume (VL) of Halstead would suggest that individual statements are quite complex. Regarding Halstead's vocabulary (V), it gives a clue about the complexity of the statements. This measure highlights whether a small number of operators are used repeatedly (which means less complexity) or if a large number of different operators are used, which will inevitably be more complex. The volume of Halstead (VL) uses length and vocabulary to give a measure of the amount of written code. The Halstead difficulty (D) uses a formula to assess complexity based on the number of unique operators and operands. It suggests how difficult is to write and maintain the code. Finally, the Halstead effort (E) will be given by the proportional measure of both difficulty (D) and volume (VL).

The values obtained for each measure in each platform can be seen in Table VI. In this table, we have averaged the results obtained for each implementation in different tested platforms, namely, OpenCL (Xeon CPU, GTX1080 GPU, and Xeon Phi), OpenMP (Xeon CPU, Xeon Phi CPU, and Odroid), and CUDA (GTX1080 GPU). In terms of coding effort, we obtain the maximum score for OpenCL, mainly due to its larger VL and D scores than their counterparts (CUDA and OpenMP). On the one hand, the VL score using OpenCL represents an increment of 53% versus CUDA and 68% versus OpenMP, respectively. On the other hand, the D score represents an enlargement of 59% versus CUDA and 20% versus OpenMP, respectively. The aforementioned values have a global impact of an overcost effort in terms of OpenCL of 52% and 59% as compared with CUDA and OpenMP, reciprocally.

We can construct a simple compromise metric taking into account the performance released by the effort needed (Mpixels/Watt per Effort). Accordingly, we present in Table VII, the obtained values after averaging the three stimuli used. Focusing on Xeon CPU and GPU GTX1080 platforms, OpenCL ratio represents 25% of their OpenMP and CUDA counterparts. This percentage raises up to 44% and 56% for Odroid and Xeon Phi devices, respectively.

Based on these preliminary results, we take into account all platforms and programming languages used in this paper in order to carry out a global analysis of the effort-performance tradeoff. Therefore, only the complete graph of all the boards studied in this work and their possible implementations through the parallel programming paradigms mentioned above are considered. We depict in Figs. 7–9 the performance-power

TABLE VII

COMPARATIVE PERFORMANCE-POWER CONSUMPTION BY EFFORT NEEDED (MPIXELS/WATT PER EFFORT IN E10 UNITS) BETWEEN DIFFERENT PLATFORMS AND DATA SETS

Platforms	P.P.L.	HYDICE	AVIRIS Cuprite	Synthetic
Xeon	OpenCL	1.18	4.60	3.46
	OpenMP	7.16	18.61	10.12
GPU	OpenCL	4.46	9.87	8.37
	CUDA	37.90	30.46	21.56
Xeon_Phi	OpenCL	0.86	5.46	2.73
	OpenMP	1.34	7.16	7.64
Odroid	OpenCL	27.30	17.56	–
	OpenMP	115.22	5.27	30.36

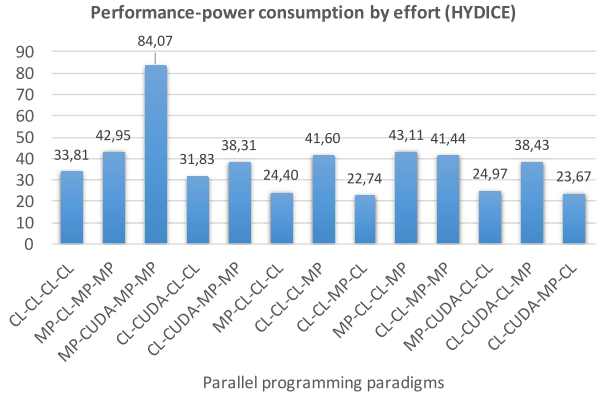


Fig. 7. Tradeoff between performance achieved, power consumption, and coding effort using different programming paradigms (Hydice stimuli).

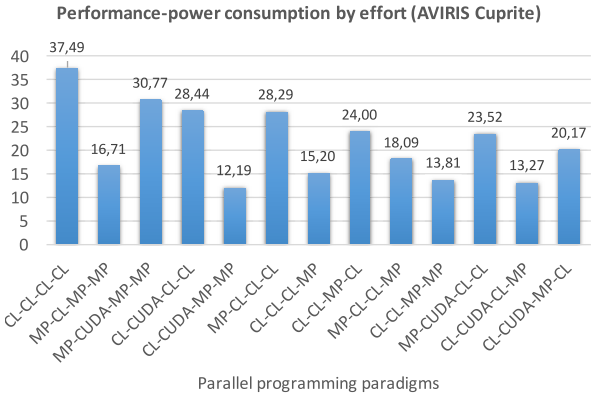


Fig. 8. Tradeoff between performance achieved, power consumption, and coding effort using different programming paradigms (Cuprite stimuli).

consumption by effort relationship using several programming paradigms, where "CL", "MP," and "CUDA" mean OpenCl, OpenMP, and CUDA programming paradigms, respectively. The real stimuli used at Fig. 7 come from the Hydice scene, whereas in Fig. 8, the stimuli come from the AVIRIS Cuprite scene. The platforms used in our experiments are represented by means of the four-tuple (Xeon CPU—GTX1080 GPU—Xeon Phi CPU Odroid board). For example, the four-tupler (MP-CUDA-CL-CL) implies the implementation in the following platforms: Xeon CPU (OpenMP), GTX1080 GPU (CUDA), Xeon Phi CPU (OpenCL), and Odroid platform (OpenCL). Regarding Fig. 9, where synthetic stimuli are used, note that we do not consider the Odroid board due to the lack of memory to deal with these stimuli. Therefore, in this case,

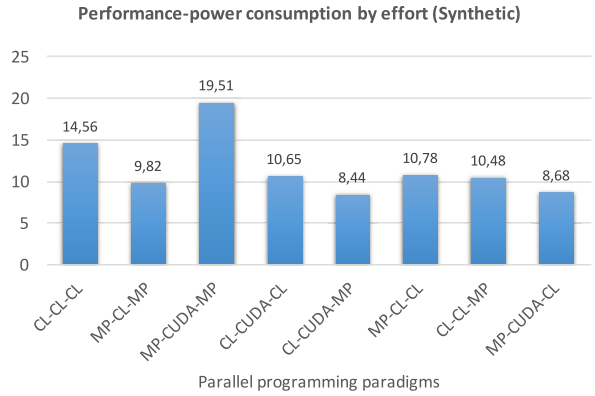


Fig. 9. Tradeoff between performance achieved, power consumption, and coding effort using different programming paradigms (synthetic image stimuli).

the platforms analyzed are established by the following three-tuple (Xeon CPU, GTX1080 GPU, and Xeon Phi CPU).

As discussed in Section IV-D, due to the characteristics of the Hydice scenario, parallelism capabilities are not efficiently exploited. The deployment of the parallel environment overlaps the performance obtained. In other words, the overhead of the kernel launched is not compensated enough by the throughput achieved. Consequently, we do not reach the expected benefits using the OpenCL paradigm in this context. On the one hand, and even despite of this circumstance, we can get an average value of 37.79 units of performance-power consumption by effort (sd=15.31). On the other hand, the majority of OpenCL-based elements (CL element number ≥ 3) provide a ratio of 28.20 units, which does not reach the average values. The four-tuple based on OpenMP and CUDA coding languages (MP-CUDA-MP-MP) contributes with the highest score (84.07 units), which appears to be a convenient solution for moderate size images.

Regarding the Cuprite scenario, the considered tuples present an average baseline of 21.68 units (sd=7.82). The solely OpenCL-based tuple (CL-CL-CL-CL) achieves the highest score (37.49 units), while other CL majority-based implementations (CL element number ≥ 3) result in a medium score of 29.55 units, upgrading the baseline ratio. This means that the use of OpenCL brings a promising relationship score when dealing with medium-sized images. In addition, the quad (MP-CUDA-MP-MP) again contributes with a high value (84.07 units), which still remains as a feasible scheme when using medium-sized images.

Finally, the synthetic scenario results in a mean value of 11.37 units (sd=3.11). The only OpenCL-based solution (CL-CL-CL) outperforms the aforementioned average, reaching up to the second highest score ratio (14.55 units). In addition, the tuples with at least two CL elements show an average ratio of 11.61 units, still over the baseline.

From the experimental stimuli studied, we can conclude that the four-tuple (CL-CL-CL-CL) is a serious candidate to obtain the efficient parallel code, providing the best relationship between performance, power consumption, and effort in designing the code. In addition, this tuple represents the

second-best choice when using large images. Concerning small images, it would not be a recommended choice due to the low performance compared to the OpenMP counterpart.

V. Conclusion

A holistic portable OpenCL implementation of the ATDCA algorithm has been presented and discussed in terms of performance, energy consumption, and code effort design considerations. The proposed implementation is compared to hand-tuned versions developed in previous works. Our results indicate that the proposed OpenCL implementation can almost overtake the performance of the hand-tuned ones on a wide variety of HPC devices, achieving real-time processing in all of them. It can also be observed that, for small data sets, the best energy efficiency (in terms of Mpixels/watt) is achieved by the ODROID platform, but when the image size is increased, GPU devices emerge as the best platform in terms of energy efficiency. Future work will focus on the development of hybrid implementations able to deal with large scenes that cannot fit into the RAM memory. We also plan on exploiting other types of architectures such as FPGAs, due to their capacity to be used as on-board processing modules with radiation tolerance in spaceborne Earth observation missions.

Acknowledgment

The authors would like to thank the Editors and the three Anonymous Reviewers for their outstanding comments and suggestions, which greatly helped them to improve the technical quality and presentation of this paper.

References

[1] A. Plaza and C.-I. Chang, *High Performance Computing in Remote Sensing*. Boca Raton, FL, USA: Taylor & Francis, 2007.

[2] Y. Tarabalka, T. V. Haavardsholm, I. Kåsen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing," *J. Real-Time Image Process.*, vol. 4, no. 3, pp. 287–300, 2009.

[3] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC decoding on multicores using OpenCL," *IEEE Signal Process. Mag.*, vol. 29, no. 4, pp. 81–109, Apr. 2012.

[4] Y. Zhang, M. Sinclair II, and A. A. Chien, "Improving performance portability in OpenCL programs," in *Proc. 28th Int. Supercomput. Conf. (ISC)*, Leipzig, Germany, Jun. 2013. [Online]. Available: https://www.springer.com/gp/book/9783642387494

[5] J. F. Fabeiro, D. Andrade, and B. B. Fraguela, "Writing a performance-portable matrix multiplication," *Parallel Comput.*, vol. 52, pp. 65–77, Feb. 2016.

[6] H. Ren and C.-I. Chang, "Automatic spectral target recognition in hyperspectral imagery," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 39, no. 4, pp. 1232–1249, Oct. 2003.

[7] 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 J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 6, no. 3, pp. 1386–1398, Jun. 2013.

[8] S. Bernabe, S. Lopez, A. Plaza, and R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis," *IEEE Geosci. Remote Sens. Lett.*, vol. 10, no. 2, pp. 221–225, Mar. 2013.

[9] 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 *Proc. IEEE 17th Int. Conf. Parallel Distrib. Syst.*, Dec. 2011, pp. 1010–1015.

[10] S. Bernabe, F. D. Igual, G. Botella, C. Garcia, M. Prieto-Matias, and A. Plaza, "Performance portability study of an automatic target detection and classification algorithm for hyperspectral image analysis using OpenCL," *Proc. SPIE*, vol. 9646, Oct. 2015, Art. no. 96460M.

[11] S. Barrachina *et al.*, "An integrated framework for power-performance analysis of parallel scientific workloads," in *Proc. 3rd Int. Conf. Smart Grids, Green Commun. IT Energy-Aware Technol.*, 2013, pp. 114–119.

[12] J. C. Harsanyi and C.-I. Chang, "Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach," *IEEE Trans. Geosci. Remote Sens.*, vol. 32, no. 4, pp. 779–785, Jul. 1994.

[13] S. Lopez, P. Horstrand, G. M. Callico, J. F. Lopez, and R. Sarmiento, "A low-computational-complexity algorithm for hyperspectral endmember extraction: Modified vertex component analysis," *IEEE Geosci. Remote Sens. Lett.*, vol. 9, no. 3, pp. 502–506, May 2012.

[14] M. Song and C. I. Chang, "A theory of recursive orthogonal subspace projection for hyperspectral imaging," *IEEE Trans. Geosci. Remote Sens.*, vol. 53, no. 6, pp. 3055–3072, Jun. 2015.

[15] C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. New York, NY, USA: Kluwer Academic, 2003.

[16] G. S. Miller, "The definition and rendering of terrain maps," *ACM SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 39–48, Aug. 1986.

[17] C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 3, pp. 608–619, Mar. 2004.

[18] J. M. Bioucas-Dias and J. M. P. Nascimento, "Hyperspectral subspace identification," *IEEE Trans. Geosci. Remote Sensing*, vol. 46, no. 8, pp. 2435–2445, Aug. 2008.

[19] H. A. Jensen and K. Vairavan, "An experimental study of software metrics for real-time software," *IEEE Trans. Softw. Eng.*, vols. SE–11, no. 2, pp. 231–234, Feb. 1985.

[20] A. H. Dutoit and B. Bruegge, "Communication metrics for software development," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 615–628, Aug. 1998. doi: 10.1109/32.707697.

[21] Z. Chang, R. Song, and Y. Sun, "Validating Halstead metrics for scratch program using process data," in *Proc. IEEE Int. Conf. Consum. Electron.-Taiwan (ICCE-TW)*, May 2018, pp. 1–5.

Sergio Bernabé received the degree in computer engineering and the M.Sc. degree in computer engineering from the University of Extremadura, Cáceres, Spain, in 2010, and the joint Ph.D. degree from the University of Iceland, Reykjavík, Iceland, and the University of Extremadura, Badajoz, Spain, in 2014.

He has been a Visiting Researcher with the Institute for Applied Microelectronics, University of Las Palmas de Gran Canaria, Las Palmas, Spain, and also with the Computer Vision Laboratory, Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil. He was a Post-Doctoral Researcher (funded by FCT) with the Instituto Superior Técnico, Technical University of Lisbon, Lisbon, Portugal, and a Post-Doctoral Researcher (funded by the Spanish Ministry of Economy and Competitiveness) with the Complutense University of Madrid (UCM), Madrid, Spain. He is currently an Assistant Professor with the Department of Computer Architecture and Automation, UCM. His research interests include the development and efficient processing of parallel techniques for different types of high-performance computing architectures.

Dr. Bernabé was a recipient of the Best Paper Award of the IEEE Journal of Selected Topics in Applied Earth Observation and Remote Sensing (JSTARS) in 2013 and the Best Ph.D. Dissertation Award at the University of Extremadura, Cáceres, in 2015. He is an Active Reviewer of international conferences and international journals, including the IEEE JSTARS, the IEEE Transactions on Geoscience and Remote Sensing (TGRS), and IEEE Geoscience and Remote Sensing Letters (GRSL).

Carlos García received the B.S. degree in physics and the Ph.D. degree in computer science from the Universidad Complutense de Madrid (UCM), Madrid, Spain, in 1999 and 2007, respectively.

He is currently an Associate Professor with UCM. He has authored more than 50 research papers, including more than 25 peer-reviewed articles in international journals. He has frequently served as a referee for international journals on image processing and high-performance computing. His research interests include high-performance computing for heterogeneous parallel architecture, including efficient parallel exploitation on modern devices such as multicore, manycore, GPUs, and FPGAs. These aspects have motivated the adoption of these technologies for the acceleration of remote sensing and image reconstruction.

Francisco D. Igual received the bachelor's and Ph.D. degrees in computer science from Jaume I University, Castellón, Spain, in 2006 and 2011, respectively.

In 2012, he joined the Computer Architecture and Automation Department, Universidad Complutense de Madrid (UCM), Madrid, Spain, as a Post-Doctoral Researcher, where he is currently an Assistant Professor. He has authored or coauthored more than 50 papers in international conferences and journals. His research interests include parallel algorithms for numerical linear algebra and multimedia applications, task scheduling, and runtime implementations on high-performance heterogenous architectures.

Guillermo Botella received the M.A.Sc. degree in physics, the M.A.Sc. degree in electronic engineering, and the Ph.D. degree in computer engineering from the University of Granada, Granada, Spain, in 1999, 2001, and 2007, respectively.

He was a Research Fellow with the Department of Architecture and Computer Technology, Universidad de Granada, funded by EU, and the Vision Research Laboratory, University College London, London, U.K. He joined the Department of Computer Architecture and Automation, Complutense University of Madrid, Madrid, Spain, as an Assistant Professor. From 2008 to 2012, he was a Visiting Professor with the Department of Electrical and Computer Engineering, Florida State University, Tallahassee, FL, USA. His research interests include digital signal processing for very large-scale integration, field-programmable gate arrays, GPGPUs, vision algorithms, and IP protection of digital systems.

Manuel Prieto-Matias received the Ph.D. degree in computer science from the Complutense University of Madrid (UCM), Madrid, Spain, in 2000.

He is currently a Full Professor with the Department of Computer Architecture, UCM. He has co-written numerous articles in journals and for international conferences in the field of parallel computing and computer architecture. His research interests include parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and on complexity-effective microarchitecture design. His research addresses emerging issues related to heterogeneous systems, memory hierarchy performance, and energy-aware computing, with a special emphasis on the interaction between the system software and the underlying architecture.

Dr. Prieto is a member of the ACM and IEEE Computer Society.

Antonio Plaza (F'15) received the M.Sc. and Ph.D. degrees in computer engineering with the Department of Technology of Computers and Communications, University of Extremadura, Badajoz, Spain, in 1999 and 2002, respectively.

He is currently the Head of the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura. He has authored more than 600 publications, including 249 JCR journal papers (more than 160 in IEEE journals), 24 book chapters, and more than 300 peer-reviewed conference proceeding papers. He has guest edited 10 special issues on hyperspectral remote sensing for different journals. His research interests include hyperspectral data processing and parallel computing of remote sensing data.

Dr. Plaza is a fellow of the IEEE for contributions to hyperspectral data processing and parallel computing of Earth observation data. He was a member of the Editorial Board of the IEEE GEOSCIENCE AND REMOTE SENSING NEWSLETTER from 2011 to 2012 and the *IEEE Geoscience and Remote Sensing Magazine* in 2013. He was also a member of the steering committee of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING (JSTARS). He was a recipient of the recognition of Best Reviewers of IEEE GEOSCIENCE AND REMOTE SENSING LETTERS in 2009, the recognition of Best Reviewers of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING in 2010, the Best Column Award of the *IEEE Signal Processing Magazine* in 2015, the 2013 Best Paper Award of the IEEE JSTARS, the most highly cited paper in the *Journal of Parallel and Distributed Computing* from 2005 to 2010, and best paper awards at the IEEE International Conference on Space Technology and the IEEE Symposium on Signal Processing and Information Technology. He served as the Director of Education Activities for the IEEE Geoscience and Remote Sensing Society (GRSS) from 2011 to 2012 and as the President of the Spanish Chapter for the IEEE GRSS from 2012 to 2016. He was an Associate Editor of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING from 2007 to 2012 and the IEEE ACCESS. He served as the Editor-in-Chief for the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING from 2013 to 2017. He is included in the 2018 Highly Cited Researchers List (Clarivate Analytics). He has reviewed more than 500 manuscripts for more than 50 different journals.