J Supercomput (2015) 71:1893–1906
DOI 10.1007/s11227-014-1372-x

Exploring the performance–power–energy balance of low-power multicore and manycore architectures for anomaly detection in remote sensing

G. León · J. M. Molero · E. M. Garzón · I. García · A. Plaza · E. S. Quintana-Ortí

Published online: 30 December 2014

Abstract In this paper, we perform an experimental study of the interactions between execution time (i.e., performance), power, and energy that occur in modern low-power architectures when executing the RX algorithm for detecting anomalies in hyperspec-

This work has been funded by Grants from the Spanish Ministry of Science and Innovation (TIN2008-01117, TIN2011-23283, TIN2012-37483-C03-01/03 and AYA2011-29334-C02-02), Junta de Andalucia (P10-TIC-6002, P11-TIC7176, P12-TIC-301) and Junta de Extremadura (PRI09A110 and GR10035) in part financed by the European Regional Development Fund (ERDF). Moreover, this work has been developed in the framework of the network High Performance Computing on Heterogeneous Parallel Architectures (CAPAP-H4), supported by the Spanish Ministry of Science and Innovation (TIN2011-15734-E).

G. León · E. S. Quintana-Ortí
Department of Engineering and Computer Science, University Jaime I, Castellón de la Plana, Spain
e-mail: leon@icc.uji.es

E. S. Quintana-Ortí
e-mail: quintana@icc.uji.es

J. M. Molero · E. M. Garzón (✉)
Department of Informatics and Agrifood Campus of International Excellence (CEIA3), University of Almería, Ctra. Sacramento s/n, 04120 Almería, Spain
e-mail: gmartin@ual.es

J. M. Molero
e-mail: jmp384@ual.es

I. García
Department Computer Architecture, Malaga University, Málaga, Spain
e-mail: igarcia@uma.es

A. Plaza
Hyperspectral Computing Laboratory, University of Extremadura, Avda. de la Universidad S/N, 10071 Cáceres, Spain
e-mail: aplaza@unex.es

tral images (i.e., signatures which are spectrally different from their surrounding data). We believe this is important because, for airborne and spaceborne remote sensing missions, power and/or energy can be in practice as relevant as performance. In this sense, this paper investigates whether several recent low-power multithreaded architectures, from ARM and NVIDIA, can be a practical alternative in this domain to a standard high-performance multicore processor, using the RX anomaly detector as a case study.

Keywords Anomaly detection · Remote sensing · Power wall · High performance · Multicore processors · Low-power architectures

1 Introduction

Anomaly detection [29] is a crucial task for the exploitation of hyperspectral data. This type of information can be viewed as a stack of images, where each image (or spectral band) represents a wavelength of the electromagnetic spectrum and each pixel has an associated spectral signature [6,11,14].[1] In this scenario, the objective for the anomaly detector is to identify spectral signatures which are spectrally distinct from their surroundings without prior knowledge [7,30]. In this context, the anomalies correspond to a set of isolated pixels with anomalous signatures (when compared to the image background), they represent a very small piece of the full image, and they only occur in the image with low probabilities [3,10,18].

Remote sensing missions are frequently performed onboard airborne devices and satellites, which may impose severe constraints on the power and energy consumption (e.g., due to battery life time or electricity being produced by the attached solar panels). The combination of fine spectral resolution, extensive earth coverage, and high dimensionality of hyperspectral images justifies the exploration of high-performance yet low-power technologies together with energy-aware novel computational algorithms that can produce a response in real time or near real time while minimizing the power/energy usage [25,28].

In this paper, we assess the potential of current low-power multicore and many-core architectures for remote sensing from the perspectives of both performance and power/energy efficiency. For this purpose, we target the RX anomaly detector developed by Reed and Yu [26]. This specific method has been applied with great success in many different hyperspectral imaging applications, being currently adopted as a standard benchmark for anomaly detection purposes [5]. The target systems for our study include two types of low-power architectures. On the one hand, we evaluate multicore general processors from Intel (Atom S1260 with 2 cores) and ARM (Cortex-A7, Cortex-A9, Cortex-A15, all three with 4 cores each) designed for low-power consumption and which are heavily employed in embedded and mobile devices. On the other hand, we consider two low-power CUDA-compatible graphics processing units (GPUs) from NVIDIA (Quadro 1000M, with 96 CUDA cores; and GK20A, with

[1] A spectral signature, or fingerprint, is the specific combination of emitted, reflected or absorbed electromagnetic radiation at varying wavelengths which can be leveraged to uniquely identify an object.

"Kepler" architecture and 192 CUDA cores). For reference, we also include in the comparison an Intel 8-core Xeon ("Sandy-Bridge") processor.

In summary, the major contribution of our paper is in the performance–power–energy evaluation of the RX detector, a representative operation for remote sensing, using a complete collection of up-to-date low-power architectures. We emphasize that the RX detector is composed of basic dense linear algebra operations (matrix–matrix products, vector operations, matrix factorizations) which are also present in other remote sensing algorithms such as, e.g., estimation of the number of endmembers, dimensionality reduction, endmember extraction or abundance estimation [2]. In consequence, we believe that the results from our experimental analysis carry beyond the RX detector, and are extensible to many other remote sensing algorithms. As an additional contribution, we introduce a simple modification of the routine that computes the correlation matrix reducing the cost of this stage roughly by half.

The rest of the paper is organized as follows. In Sect. 2, we offer a short description of related work. In Sect. 3, we briefly review the RX algorithm for anomaly detection, and in Sect. 4 we offer some details on the parallelization of this algorithm. The experimental evaluation of the RX algorithm on the target architectures using a real hyperspectral image follows in Sect. 5. Finally, the paper is closed with a discussion of concluding remarks in Sect. 6.

2 Related work

The RX detector has been implemented and optimized for a variety of high-performance architectures in previous works, including clusters, multicore architectures, and manycore GPUs [20,22,24]. In all these cases, the focus was on raw performance and no analysis of the power-energy consumption was made. A few recent studies shed some light into the interactions of the performance–power–energy triangle using remote sensing algorithms to analyze hyperspectral data. In Castillo et al. [4], the authors offer a study similar to ours, using parts of a spectral unmixing chain, but focus on a specialized multicore DSP (digital signal processor) which is completely different from the point of view of programming to the low-power architectures that we consider in this paper. That study also included quite an old Intel Atom (D510 with 2 cores, from 2010) as well as an old ARM Cortex-A9, with 2 cores only. In [27,28], the authors presented complementary analyses of the time–power–energy balance in remote sensing on general-purpose multicore processors and high-performance GPUs, respectively. The first paper only considered an extreme platform equipped with four 12-core AMD processors (Opteron 6,172). In the second one, the target GPUs comprised two high-performance but power-hungry boards such as the Tesla K20c ("Kepler") and the GTX 480 ("Fermi"), with 2,496 and 240 CUDA cores, respectively. The study also included the old Quadro 1,000 M that we utilize in our experiments.

It is also worth mentioning that FPGAs have been shown to be a competitive low-power architecture in many remote sensing operations (see, among many others, [1,13,15]), which can also be applied to anomaly detection. However, for this particular paper, we preferred to focus on more conventional multithreaded technologies. The reason that motivated our choice is that, compared with FPGA, multithreaded

processors are programmed via high-level languages, which are more amenable to scientists and lead to less fatiguing implementations.

3 The RX algorithm

The RX anomaly detector has been widely used in hyperspectral signal and image processing [26]. A practical variant of the RX algorithm replaces the covariance matrix by the sample correlation matrix and removes the mean vector from each b-dimensional hyperspectral pixel (column) vector

$$\mathbf{x} = \left[x^0, x^1, \ldots, x^b\right]^{\mathrm{T}} \in \mathbb{R}^b$$

where b is related to the number of bands of the image [7]. This variation represents an adaptation of the original RX algorithm to online analysis scenarios which does not impair its ability to detect anomalies.

The RX filter is defined as

$$\delta(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} R^{-1} \mathbf{x}, \tag{1}$$

and the correlation matrix R is given by

$$R = \frac{H^{\mathrm{T}} \cdot H}{n}, \tag{2}$$

where the transposed image, $H^{\mathrm{T}} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n] \in \mathbb{R}^{b \times n}$, consists of $n = l \times s$ pixel vectors; and l and s (the lines and samples, respectively) define the spatial dimensions of the image H [6,26]. The results from the anomaly detection filter themselves can be represented as a grayscale image, where each value represents the probability of the associated pixel being anomalous.

Two well-defined stages can be identified in the computation of RX:

1. Calculation of the correlation matrix R: this initial stage requires the computation of the matrix–matrix product in Eq. (2). In principle, this operation has a computational complexity of $2b^2 \cdot n$ floating-point arithmetic operations (or flops, for short). For real hyperspectral images, though, this cost can be reduced by taking into account the symmetry of the result (the correlation matrix). Concretely, because of this property, it is not necessary to compute the entire matrix but only its lower or upper triangle. Therefore, the computational complexity for this stage can be halved, to approximately $b^2 \cdot n$ flops only, compared with the prior implementation in Molero et al. [21]. Additionally, the execution time for this stage is in practice short as the matrix–matrix multiplication is implemented as an extremely tuned kernel for most architectures.
2. Computation of the filter $\delta(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} R^{-1} \mathbf{x}$: the classic implementation of this stage first computes the inverse (or pseudoinverse) matrix $Z = R^{-1}$ explicitly. This is next followed by the evaluation, for each pixel of the image, of the expression

$$\delta(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} Z \mathbf{x}, \tag{3}$$

which can be implemented as a matrix-vector product ($\mathbf{y}(\mathbf{x}) = Z\mathbf{x}$), followed by an inner (or dot) product $\left(\delta(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{y}(\mathbf{x})\right)$ [18,22].
Nevertheless, a more clever approach that relies on a specialized factorization of R can contribute to a reduction in the cost of this operation [20]. In particular, given that the correlation matrix R is square, symmetric and positive definite, an efficient way to proceed is to initially compute the Cholesky factorization of this matrix:

$$R = U^{\mathrm{T}} U, \tag{4}$$

where the upper triangular matrix $U \in \mathbb{R}^{b \times b}$ is referred to as the Cholesky factor of R [12]. This is then simply followed by the calculation:

$$\delta(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} R^{-1} \mathbf{x} = \left(U^{-\mathrm{T}}\mathbf{x}\right)^{\mathrm{T}} \left(U^{-\mathrm{T}}\mathbf{x}\right) \equiv \mathbf{z}(\mathbf{x})^{\mathrm{T}}\mathbf{z}(\mathbf{x}), \tag{5}$$

for each pixel of the image. In summary, this alternative method, based on the Cholesky factorization, consists of the following three steps:

(a) Decompose the correlation matrix R into a product of two triangular matrices $R = U^{\mathrm{T}}U$ ($b^3/3$ flops).
(b) Solve the triangular system $U^{\mathrm{T}}\mathbf{z}(\mathbf{x}) = \mathbf{x}$ for the intermediate vector $\mathbf{z}(\mathbf{x})$ (b^2 flops per pixel).
(c) Finally, compute the inner product $\delta(\mathbf{x}) = \mathbf{z}(\mathbf{x})^{\mathrm{T}}\mathbf{z}(\mathbf{x})$ ($2b$ flops per pixel).

Proceeding in this structure-aware manner, the computational cost of this stage is reduced to $\frac{b^3}{3} + b^2 \cdot n + 2b \cdot n$ flops, which represents a significant lower amount of computations compared to the use of the classic method [21].

4 Parallelization of the RX algorithm

This work is focused on the performance–power–energy evaluation of the RX algorithm on modern low-power representants of multicore and manycore GPU architectures. For this purpose, we have leveraged two parallel RX codes tuned, respectively, for multicore processors and GPUs. These parallel RX algorithms were developed as part of previous work [19–21], where they were evaluated on high-performance multicore and graphics accelerators. The versions that we employ in this paper to extract performance of the low-power processors are slight variants of the original algorithms. We next describe the main properties of both parallel RX implementations.

4.1 Parallel RX detector on multicore processors

Algorithm 1 sketches the multicore implementation of the RX algorithm. This code takes advantage of the underlying hardware parallelism via OpenMP threads and multithreaded implementations of dense linear algebra libraries such as LAPACK and

BLAS [8].[2] We note that this functionality has been optimized for different multicore architectures to exploit the inherent parallelism of the algebraic computations [9,16,17].

In the RX algorithm, the calculation of the correlation matrices (Stage 1) is performed via the BLAS routine `dsyrk` (symmetric rank-k update) to exploit the symmetry of the result. For the factorization of the correlation matrix (Stage 2), we employed routine `dpotrf` from LAPACK; and routine `dtrtrs`, from BLAS, is applied to obtain the intermediate vector(s) $\mathbf{z}(\mathbf{x})$ as a triangular linear system solve with multiple right-hand sides (one right-hand side per pixel). Finally, the inner product involving $\mathbf{z}(\mathbf{x})$ is implemented via the BLAS kernel `ddot`, with the loop around it parallelized using OpenMP threads.

Algorithm 1 Pseudocode of the multicore implementation of the RX algorithm.

$H = Load\ Hyperspectral\ Image$
$R = \frac{H^{\mathrm{T}} \cdot H}{l \cdot s}$ {Matrix–matrix product using `dsyrk` function}
$U = Cholesky\ factorization\ (R)$ {Cholesky fact. of R matrix using `dpotrf` function}
$z = Solve\ linear\ systems\ (U, H)$ {Solve triangular linear system with multiple right-hand sides using `dtrtrs` routine}
for $j = 1$ **to** $l \cdot s$ **do** {Loop parallelized via OpenMP threads}
  $\delta_j = \mathbf{z}_j^{\mathrm{T}} \cdot \mathbf{z}_j$ {Compute RX filter of jth pixel using `ddot` kernel}
end for

4.2 Parallel RX on GPUs

In this case, the stages of the RX algorithm have been parallelized using different strategies. In the first stage, the correlation matrix is computed using the entire GPU architecture. In this case, the calculation of the correlation matrix is cast in terms of a matrix–matrix product, computed via kernel `dsyrk` from the CUBLAS library (available as part of the NVIDIA CUDA SDK [23]). For real hyperspectral images, the matrix–matrix product is usually large enough to be efficiently performed in the GPU, since the two matrices being multiplied are of dimension $b \times n$ and $n \times b$.

For the second stage, we leveraged tailored kernels for the Cholesky factorization, the solution of the triangular linear system, and the inner product from [21]. In this case, the small size of the system led us to implement our own algebraic kernels instead of leveraging routines from existing general libraries. Specifically, the Cholesky factorization of the matrix is computed using a single CUDA block, to avoid communication penalties, and the results are stored in the GPU memory.

For the solution of the linear system, each pixel of the image is assigned to one CUDA block, and each band of the image is mapped onto one CUDA thread. Proceeding in this manner, the threads of a block cooperate in the solution of a single independent triangular linear system, while all the blocks work independently in par-

[2] http://www.netlib.org/lapack.

allel, avoiding communications between different blocks. Finally, the inner product involving $\mathbf{z}(\mathbf{x})$ is merged into the same kernel with the system solver.

5 Experimental results

This section is opened with a technical description of the experimental hardware setup, consisting of the multicore architectures and power measurement devices. This is followed by a brief review of the hyperspectral data tested that was employed in all the experiments. Finally, the section is concluded with the analysis of the performance–power–energy balance of the RX algorithm, when applied to detect anomalies of the reference hyperspectral scene on the target architectures.

5.1 Hardware setup

The experimental study was performed on a variety of stand-alone computing "platforms" or boards, and a conventional server, all of them equipped with recent processor technology from ARM, Intel and NVIDIA; see Table 1. The results with the A9 where simply collected from the ARM processor in the Carma development board.

These systems represent three current trends in the design of processors:

– The ARM Cortex and Intel Atom are lower power general-purpose processors, with big (or "fat") cores offering a functionality similar to their server relatives

Table 1 Main features of low-power multicore processors, GPU-equipped boards, and the Intel server (top, middle and bottom, respectively) employed in the experiments

Acronym	Architecture	#Cores	Frequency (GHz)	Idle power (W)	RAM (GB)
	Exynos5 Octa			3.1	2
A15	ARM Cortex-A15	4	1.2, 1.6		
A7	+ARM Cortex-A7	+4	0.25, 0.45, 0.6		
IAT	Intel Atom S1260	2	0.6, 1.0, 1.5, 2.0	40.3	8
	Carma dev. kit			10.2	
QDR	NVIDIA Quadro 1000M	96	1.4		2
A9	+ARM Cortex-A9	4	0.76, 1.0, 1.3		2
	Jetson TK1 dev. kit			11.8	
KEP	NVIDIA Kepler	192	0.85		1.7
–	+ARM Cortex-A15	4	2.32		16
ISB	Intel Xeon i7-3930 (Sandy-Bridge)	6	1.2, 1.9, 2.5, 3.2	95.5	24

Most of these architectures can operate at different frequencies, which can be set by the user. The column labeled as "frequency" lists the values that were employed in the experiments. In all cases, the "idle power" is measured with the platform idle during 30 s, and with the processors set at the nominal frequency. For the GPU-equipped boards (Carma and Jetson kits), the idle power includes the consumption of both the processor and the accelerator

from Intel (e.g., Xeon, i3, i5, i7 series) and AMD (Opteron series), but much lower power dissipation and, obviously, lower performance as well.
- The "small" GPUs from NVIDIA in the Carma development kit and the Jetson TK1 embedded development kit feature high hardware parallelism (almost 100 cores in QDR and close to 200 in KEP), very appropriate for data-parallel operations. On the other hand, they achieve these high numbers at the expense of embedding fine-grained CUDA cores into the chip, which are little more than mere floating-point arithmetic units. In consequence, these GPUs cannot operate alone, but are instead integrated into a low-power board, in both cases together with a general-purpose processor from ARM in charge of the control.
- The Intel Xeon is a modern general-purpose processor integrated into a complete server node, with a disk, as well as two network cards and several PCI-e connectors embedded into the motherboard.

The following operating system+compiler were used in each platform:

- Exynos5 Octa: `Odroid 3.4.75+gcc 4.8.1`;
- IAT: `CentOS 6.5 (2.6.32)+icc 11.1`;
- Carma Development kit: `Tegra-Ubuntu 3.1 armv7+gcc 4.6.3+nvcc 5.0`;
- Jetson TK1 embedded development kit : `Tegra-Ubuntu 3.1 armv7 +gcc 4.8.2+nvcc 6.0.1`;
- ISB: `Rocks 2.6.32+icc 12.1.3`.

Nevertheless, as most of the operations of the RX algorithm are cast in terms of the kernels available in these highly tuned libraries, the operating system and compiler that was employed in these cases have a negligible contribution. Tuned implementations of the computational kernels that appear in the RX algorithm were obtained, for the Intel-based platforms and the NVIDIA GPUs, respectively, from recent releases of Intel MKL (release 11.1 for IAT and 12.1 for ISB) and NVIDIA CUBLAS (release 5.0 for QDR and 6.0 for KEP). For the ARM, we relied on BLIS (release 0.1.2–4) for the specialized matrix–matrix product that appears in the first stage of the algorithm (see Eq. 2),[3] and ATLAS (release 3.10.2) for the remaining computations.[4]

To measure power, we employed a WattsUp? Pro .Net wattmeter. This device is plugged to the cable that connects the electrical socket to the system, and reports total external AC power, with a sampling rate of 1 Hz, an accuracy of ± 1 %, and a resolution of 0.1 W. We warmed up the platforms by executing the RX algorithm repeatedly during 3 min before the sampling was initiated. Power measures were then continuously recorded while the test (i.e., the algorithm) was run during three additional minutes. Power was then averaged over this period and multiplied by the execution time of a single instance of the algorithm to obtain its (total) energy consumption. A complementary metric to compare the energy efficiency of the different platforms is the net energy consumption, which was obtained by subtracting the product of idle power by the time from the energy consumption. This measure reflects more accurately the effective amount of energy necessary to perform the work, canceling the effect

[3] http://code.google.com/p/blis/.

[4] http://math-atlas.sourceforge.net/.

of unnecessary components (e.g., the disk) on the consumption. Hereafter, execution time is reported in seconds (s), power in Watts (W) and energy in Joules ($J = W \cdot s$),

5.2 Hyperspectral scene

The hyperspectral image for the experiments was collected by the AVIRIS instrument, flown by NASAs Jet Propulsion Laboratory over the World Trade Center area in New York city on September 16, 2001. The size of the full scene is 614×512 pixels, with 224 spectral bands, for a total size of about 140 MB (this is the standard size of the data chunks collected by the AVIRIS instrument before saving the data to disk in the onboard data collection). The acquisition time required by the AVIRIS sensor is 5.1 s for the entire image.

All test with this image were performed using IEEE 754 real double-precision arithmetic.

5.3 Performance–power–energy analysis

Table 2 reports the performance, power dissipation and energy consumption of the RX algorithm executed in the seven target architectures, using different number of cores and operation frequencies. For the power, we provide three values, P_{avg}, P_{max} and P_{net}, corresponding to the average, maximum and net power, respectively. The last metric is obtained by subtracting the idle power from the average power for each platform. For the energy, we include the net and total results.

Consider the execution time first. Unsurprisingly, the best option is to execute the algorithm at the highest possible frequency and using all the cores of the platform. The only exception to this is A9 where, for all frequencies, it is actually more convenient to use 3 cores instead of 4. Moreover, in this particular system there is a negligible difference between the executions times observed with the processor operating at 1.0 and 1.3 GHz, when 3 cores are in use. If we compare now the performance of all the architectures, the winner is ISB with KEP quite close to it. These two systems are roughly $3\times$ faster than the low-power QDR, about one order of magnitude better than A15, and outperform A9 and IAT even by a larger margin. This is not totally unexpected, as ISB is a processor optimized for performance, while all other platforms place energy efficiency at least on par with performance. The interesting observation here is that KEP almost matches the performance of the power-hungry ISB. At this point, it is worth pointing out that whether these differences are relevant or not actually depends on the application. In particular, for certain scenarios, anomaly detection must be performed in real time, i.e., at the pace that the images are acquired. In these circumstances, an execution time above the threshold defined by these conditions is unacceptable. On the other hand, an execution time that is lower may not offer any benefit either, or can be even undesirable if, e.g., incurs into a higher power dissipation rate or increases the total energy consumption.

The results on IAT deserve some further comments. We performed some additional experiments on this platform to discover that the source of its lower performance is the lack of efficient implementations of the BLAS kernels that are part of Intel MKL for this particular architecture.

Table 2 Execution time, power and energy consumption of the RX algorithm executed on the target platforms

Architecture	Frequency (GHz)	#Cores	Time	P_{avg}	P_{max}	P_{net}	E_{net}	E_{tot}
A15	1.2	1	12.5	**6.8**	**8.1**	**3.7**	**46.3**	85.2
	1.2	2	9.1	8.2	9.1	5.1	47.4	75.8
	1.2	3	8.0	8.9	9.8	5.8	47.3	**72.3**
	1.2	4	9.2	8.8	10.1	5.7	53.0	81.6
	1.6	1	10.0	10.0	18.3	6.9	69.7	100.7
	1.6	2	7.3	12.1	19.3	9.0	66.4	89.1
	1.6	3	6.3	14.4	19.1	11.3	71.6	91.2
	1.6	4	**5.8**	15.3	17.9	12.2	70.8	88.8
A7	0.25	1	116.1	**3.1**	**3.3**	**0.1**	**9.2**	369.3
	0.25	2	82.6	3.2	3.4	0.1	12.3	268.4
	0.25	3	68.6	3.3	3.4	0.2	13.7	226.5
	0.25	4	62.5	3.3	3.4	0.2	15.0	208.8
	0.45	1	66.3	3.3	3.6	0.2	17.9	223.7
	0.45	2	45.9	3.5	3.6	0.4	19.3	161.7
	0.45	3	40.3	3.6	3.7	0.5	20.1	145.2
	0.45	4	37.1	3.6	3.8	0.5	20.4	135.5
	0.6	1	51.3	3.6	4.0	0.5	26.1	185.3
	0.6	2	37.8	3.8	4.1	0.7	28.7	146.2
	0.6	3	30.6	4.0	4.3	0.9	27.5	122.4
	0.6	4	**27.2**	4.1	4.5	1.0	28.5	**112.9**
IAT	0.6	1	234.0	**41.2**	**41.4**	**0.9**	212.9	9,643.5
	0.6	2	118.7	41.4	41.6	1.1	137.7	4,922.1
	1.0	1	140.6	41.5	41.7	1.2	177.2	5,845.2
	1.0	2	71.2	42.1	42.3	1.8	128.2	2,998.9
	1.5	1	94.5	42.0	42.3	1.7	165.4	3,974.3
	1.5	2	48.0	42.8	43.0	2.5	120.5	2,056.6
	2.0	1	70.6	42.5	42.7	2.2	158.1	3,003.3
	2.0	2	**35.9**	43.6	43.8	3.3	**120.0**	**1,569.0**
QDR	1.4	4	**1.7**	**17.0**	**17.1**	**6.7**	**11.8**	**29.8**
A9	0.76	1	45.5	**11.6**	**13.0**	**1.3**	**63.3**	530.6
	0.76	2	32.5	12.5	13.2	2.2	73.9	407.7
	0.76	3	39.5	12.9	14.0	2.7	107.5	513.0
	0.76	4	63.3	13.1	13.6	2.8	181.6	830.5
	1.0	1	36.0	12.1	13.6	1.8	68.1	437.8
	1.0	2	25.6	13.2	14.2	2.9	76.5	339.7
	1.0	3	**22.7**	14.0	14.6	3.8	86.7	**319.9**
	1.0	4	32.9	14.2	14.8	3.9	131.6	469.8
	1.3	1	32.6	12.5	13.7	2.2	74.1	408.7
	1.3	2	24.9	13.3	14.1	3.0	76.8	332.6

Table 2 continued

Architecture	Frequency (GHz)	#Cores	Time	P_{avg}	P_{max}	P_{net}	E_{net}	E_{tot}
	1.3	3	22.8	14.0	14.6	3.7	86.4	320.3
	1.3	4	26.4	14.4	15.1	4.1	109.7	380.7
KEP	0.85	1	**0.7**	**13.5**	**15.4**	**1.7**	**1.2**	**9.6**
ISB	1.2	1	5.0	**109.2**	**111.5**	**13.7**	69.0	549.0
	1.2	2	2.5	114.2	118.3	18.7	47.2	288.7
	1.2	4	1.3	123.6	126.1	28.1	38.6	169.6
	1.2	6	1.0	128.6	130.2	33.1	**33.5**	130.3
	1.9	1	3.1	118.9	121.5	23.4	74.8	379.8
	1.9	2	1.6	127.2	130.6	31.7	51.1	204.6
	1.9	4	0.9	141.8	144.9	46.3	42.0	128.7
	1.9	6	0.7	149.6	153.0	54.1	39.1	**108.1**
	2.5	1	2.4	128.5	130.2	33.0	81.2	316.1
	2.5	2	1.2	140.1	143.2	44.6	55.6	174.4
	2.5	4	0.7	165.3	171.0	69.8	49.5	117.3
	2.5	6	0.6	175.5	180.2	80.0	49.5	108.5
	3.2	1	1.9	141.9	144.4	46.4	90.1	275.6
	3.2	2	0.9	161.8	167.3	66.3	65.9	160.8
	3.2	4	0.5	207.6	217.2	112.1	65.1	120.5
	3.2	6	**0.5**	222.2	231.1	126.7	69.6	122.2

The values in bold face highlight, for each architecture, the best option from the point of views of performance, (average, maximum, net) power and (net, total) energy. The net values are obtained after substracting the idle power/energy from the total values

Let us discuss next the power dissipation. The average, maximum and net power have the opposite behavior to performance, with values that increase linearly with the frequency, and more irregularly with the number of cores for all three variants of the power metric. If we analyze the systems, the best platform from this point of view is A7, which only dissipates 3.1–4.1 W on average, and 3.3–4.1 W at most. Compare this, for example, with the 109.2–222.2 W drawn by ISB. Even if we subtract the idle power, to gain a better perspective of the effective (i.e., net) power that is dissipated while performing the actual work, the differences between A7 and ISB are impressive. Looking into the results of IAT we can observe that, although being characterized as a low-power processor, there is a large gap between the dissipation rates of this architecture compared with the alternative power efficient designs.

Finally, consider the energy consumption. Here the variability is high and the optimal pair (number of cores, frequency) depends on the target architecture. Focussing, for example, on the total energy, the optimal on A15 employs 3 cores and the lowest frequency. (Remember that, on this processor, the performance attained using 3 cores was actually higher than that obtained with 4 cores.) On A7 and IAT, the best option is to employ the highest number of cores and set the frequency to the highest. Finally, for A9 and ISB, the optimal is somewhat in the middle; in the ARM case, 3 cores offer

better results than 4; and in the Intel processor it is more beneficial to employ the largest number of cores. In a global comparison, if we consider the total energy, the most efficient system is KEP, with QDR in second place. They are next followed by A15, and then both A7, ISB close together. If the reference metric is the net energy, then A15 and A7 simply swap their positions. IAT suffers from the bad combination of long execution time and relatively high power dissipation rate in both net and total energy.

6 Concluding remarks

We have analyzed the performance as well as the power and energy consumption of a collection of recent low-power architectures using the RX algorithm for anomaly detection as a case study. Our experimental results demonstrate the potential of low-power GPUs to deliver reasonable performance and high power/energy gains. In addition, the low-power general-purpose processors from ARM considered in this study offer a lower performance–power ratio than the graphics accelerators, but a more familiar programming interface. In general, the performance of both types of low-power systems is far from that of a current Intel Xeon processor. Nevertheless, whether this is relevant for remote sensing depends on the application scenario, which may enforce either performance, power or energy as the primary goal. The RX detector is composed of basic dense linear algebra operations (matrix–matrix products, vector operations, matrix factorizations) which are also present in other remote sensing algorithms such as, e.g., estimation of the number of endmembers, dimensionality reduction, end-member extraction or abundance estimation. A close inspection to the list of operations that compose the RX Algorithm (see Sect. 3) reveals that all of them, except the computation of the inner products, are compute-bound kernels. On the other hand, the contribution of the inner products to the total theoretical cost is minor (e.g., one order of magnitude lower than the triangular system solve). Therefore, we can expect that the global RX Algorithm is a compute-bound method and, provided reasonably tuned implementations of the underlying kernels are leveraged, memory bandwidth will not play a major role on its performance. We believe that the results from our experimental analysis with low-power architectures carry beyond the RX detector, and they are extensible to many remote sensing algorithms and, in general, any other algorithm that can be decomposed into compute-bound kernels.

Acknowledgments We thank Francisco D. Igual, from Universidad Complutense de Madrid (Spain), for his help with the optimization and installation of BLAS for several of the platforms considered in this work. Last but not least, we gratefully thank the Associate Editor and the Anonymous Reviewers for their detailed comments and suggestions, which greatly helped us to improve the technical quality and presentation of our manuscript.

References

1. Bernabe S, López S, Plaza A, Sarmiento R, García Rodríguez P (2011) FPGA design of an automatic target generation process for hyperspectral image analysis. In: IEEE 17th international conference on parallel and distributed systems, ICPADS 2011, December 7–9, Tainan, pp 1010–1015

2. Bioucas-Dias J, Plaza A, Dobigeon N, Parente M, Du Q, Gader P, Chanussot J (2012) Hyperspectral unmixing overview: geometrical, statistical, and sparse regression-based approaches. IEEE J Sel Top Appl Earth Obs Remote Sens 5(2):354–379
3. Borghys D, Kasen I, Achard V, Perneel Ch (2012) Comparative evaluation of hyperspectral anomaly detectors in different types of background. In: Proc. SPIE, pp 83902J–83902J-12
4. Castillo MI, Fernández JC, Igual FD, Plaza A, Quintana-Ortí ES, Remon A (2014) Hyperspectral unmixing on multicore DSPs: trading off performance for energy. IEEE J Sel Top Appl Earth Obs Remote Sens 7(6):2297–2304
5. Chang C-I (2003) Hyperspectral imaging: techniques for spectral detection and classification. Kluwer Academic, Plenum Publishers, New York
6. Chang C-I (2013) Hyperspectral data processing: algorithm design and analysis. Wiley, New Jersey
7. Chang C-I, Chiang S-S (2002) Anomaly detection and classification for hyperspectral imagery. IEEE Trans Geosci Remote Sens 40(6):1314–1325
8. Dongarra JJ, Du Croz J, Hammarling S, Duff I (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17
9. Dongarra JJ, Duff IS, Sorensen DC, Der Vorst HV (1990) Solving linear systems on vector and shared memory computers. Society for Industrial and Applied Mathematics (SIAM), Philadelphia
10. Du B, Zhang L (2011) Random selection based anomaly detector for hyperspectral imagery. IEEE Trans Geosci Remote Sens 49(5):1578–1589
11. Goetz AFH, Vane G, Solomon JE, Rock BN (1985) Imaging spectrometry for earth remote sensing. Science 228:1147–1153
12. Golub GH, Van Loan CF (1996) Matrix computations, 3rd edn. The Johns Hopkins University Press, Baltimore, Maryland
13. González C, Sánchez S, Paz A, Resano J, Mozos D, Plaza A (2013) Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing. Integration 46(2):89–103
14. Green RO, Eastwood ML, Sarture CM, Chrien TG, Aronsson M, Chippendale BJ, Faust JA, Pavri BE, Chovit CJ, Solis M et al (1998) Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS). Remote Sens Environ 65(3):227–248
15. Hsueh M, Chang C (2008) Field programmable gate arrays (FPGA) for pixel purity index using blocks of skewers for endmember extraction in hyperspectral imagery. Int J High Perform Comput Appl 22(4):408–423
16. Intel (2012) Intel math kernel library–documentation. Retrieved from https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation
17. Marqués M, Quintana-Ortí G, Quintana-Ortí ES, van de Geijn R (2011) Using desktop computers to solve large-scale dense linear algebra problems. J Supercomput 58:145–150
18. Matteoli S, Diani M, Corsini G (2010) A tutorial overview of anomaly detection in hyperspectral images. IEEE Aerosp Electron Syst Mag 25(7):5–28
19. Molero JM, Garzón EM, García I, Plaza A (2012) Anomaly detection based on a parallel kernel RX algorithm for multicore platforms. J Appl Remote Sens 6(1):11. doi:10.1117/1.JRS.6.061503
20. Molero JM, Garzón EM, García I, Plaza A (2013) Analysis and optimizations of global and local versions of the RX algorithm for anomaly detection in hyperspectral data. IEEE J Sel Top Appl Earth Obs Remote Sens 6(2):801–814
21. Molero JM, Garzón EM, García I, Quintana-Ortí ES, Plaza A (2014) Efficient implementation of hyperspectral anomaly detection techniques on GPUs and multicore processors. IEEE J Sel Top Appl Earth Obs Remote Sens 7(6):2256–2266
22. Molero JM, Paz A, Garzón EM, Martínez JA, Plaza A, García I (2011) Fast anomaly detection in hyperspectral images with RX method on heterogeneous clusters. J Supercomput 58(3):411–419. doi:10.1007/s11227-011-0598-0
23. Nvidia (2014) CUBLAS library. User guide. Retrieved from http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf
24. Paz A, Plaza A, Plaza J (2009) Comparative analysis of different implementations of a parallel algorithm for automatic target detection and classification of hyperspectral images. In: Proc. SPIE, vol 7455, pp 74550X–74550X-11
25. Plaza A, Chang C-I (2008) Preface to the special issue on high performance computing for hyperspectral imaging. Int J High Perform Comput Appl 22(4):363–365
26. Reed IS, Yu X (1990) Adaptive multiple-band cfar detection of an optical pattern with unknown spectral distribution. IEEE Trans Acoust Speech Signal Process 38:1760–1770

27. Remón A, Sánchez S, Bernabé S, Quintana-Ortí ES, Plaza A (2013) Performance versus energy consumption of hyperspectral unmixing algorithms on multi-core platforms. EURASIP J Adv Signal Process 2013:68
28. Sánchez S, León G, Plaza A, Quintana-Ortí ES (2014) Assessing the performance–energy balance of graphics processors for spectral unmixing. IEEE J Sel Top Appl Earth Obs Remote Sens 7(6):2305–2316
29. Shaw G, Manolakis D (2002) Signal processing for hyperspectral image exploitation. IEEE Signal Process Mag 19:12–16
30. Stein DWJ, Beaven SG, Hoff LE, Winter EM, Schaum AP, Stocker AD (2002) Anomaly detection from hyperspectral imagery. IEEE Signal Process Mag 19:58–69