A Jungle Computing approach to common image source identification in large collections of images


DOI

Publication date
2018

Document Version
Final published version

Published in
Digital Investigation

License
CC BY

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
A Jungle Computing approach to common image source identification in large collections of images

B. van Werkhoven a, *, P. Hijma b, c, C.J.H. Jacobs c, J. Maassen a, Z.J.M.H. Geradts d, b, H.E. Bal c

a Netherlands eScience Center, Science Park 104, Amsterdam, the Netherlands
b University of Amsterdam, Science Park 904, Amsterdam, the Netherlands
c Dept. Computer Science, VU University Amsterdam, de Boelelaan 1081, Amsterdam, the Netherlands
d Netherlands Forensics Institute, Laan van Ypenburg 6, The Hague, the Netherlands

A R T I C L E   I N F O
Article history:
Received 27 March 2018
Received in revised form 2 August 2018
Accepted 9 September 2018
Available online 18 September 2018

Keywords:
Jungle computing
Digital camera identification
Large-scale common source identification
Photo-response non-uniformity
Digital forensics

A B S T R A C T
Analyzing digital images is an important investigation in forensics with the ever increasing number of images from computers and smartphones. In this article we aim to advance the state-of-the-art in common image source identification (which images originate from the same source camera). To this end, we present two types of applications for different goals that make use of a) a modern Desktop computer with a GPU and b) highly heterogeneous cluster computers with many different kinds of GPUs, something we call computing jungles. The first application targets medium-scale investigations, for example within a crime laboratory, the second application is targeted at large-scale investigations, for example within institutions. We advance the state-of-the-art by 1) explaining in detail how we obtain the performance to 2) support large databases of images in reasonable time while 3) not giving up accuracy. Moreover, we do not apply filtering ensuring that 4) our results are highly reproducible.

© 2018 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Introduction

The ubiquitousness of digital cameras, for example in smartphones, in CCTV-systems, and integrated in devices such as cars and drones, results in a rapidly growing number of images. As a result, images play a large role in forensic investigation and become increasingly more important to solve cases. Examples are child pornography and homicide cases where images can help to find links between suspects and victims.

In forensic investigations the following two related questions are often asked: whether images were made with a specific camera (Qu et al., 2013), for example owned by a suspect, and whether different images were made with the same camera (Gisolf et al., 2014). For the first investigation, called source identification, the camera has to be available, for example seized by the police, and the goal is to match the sensor of the camera to images. For the second investigation, called common source identification, the camera is not available and the goal is to find images that are made with the same camera. This entails comparing a set of images amongst each other and group the images so that images with a common source form a cluster. In contrast to source identification, common source identification is much more computationally intensive as \( \mathcal{O}(N^2) \) comparisons have to be made for \( N \) images, whereas source identification only needs \( N \) comparisons.

To answer these types of questions, the method that is used in forensic practice and casework worldwide is Photo Response Non Uniformity (PRNU) (Fridrich, 2013). Within ENFSI (the European Network of Forensic Institutes) (Ho and Li, 2015) in the past two years, proficiency tests have been organized to compare the results of the different laboratories. Several publications (Bertini et al., 2015; Karaküçük et al., 2015) apply the technology on images from social networks.

PRNU is a measure of imperfection of an image sensor, which is caused by the manufacturing process of the CCD or CMOS sensors in the camera. Many papers (Singh and Malik) have been published on methods for extracting and comparing PRNU patterns. The first approaches were based on pixel defects (Geradts et al., 2001), whereas later ones used the pattern caused by variation of...
Since common source identification (no camera available) is computationally expensive, it is common practice to trade off accuracy for performance. Examples are reducing the number of images to compare, using only a crop of the image for the noise pattern, noise patterns with loss of information, comparison methods with reduced accuracy, and clustering with approximations (Gisolf et al., 2014; Chuang et al., 2011; Fridrich, 2013; Bayram et al., 2012; Li, 2010).

However, the reduced accuracy can lead to more false positives and negatives with reduced confidence in the results. By using the PRNU with the accuracy maximized, we minimize the number of false positives. In this article, since digital images are prevalent and of high importance in a growing number of cases, we aim to advance the state-of-the-art in common source identification. Our goal is to obtain high performance and high accuracy by using the full noise patterns, offering high-quality comparisons and clustering while supporting large data sets. Our approach is to use modern compute infrastructure effectively.

Many-core hardware, such as Graphics Processing Units (GPUs), is increasingly becoming the computing platform of choice, because of the high performance against low costs and power consumption. In fact, 7 out of top 10 fastest supercomputers (P500 Supercomputer Site, 2018) (as of November 2017) use many-core accelerators.

Since the field of many-core processors witnesses a high development rate, with each year faster processors, high-performance computing systems have become more heterogeneous, often containing a variety of many-core devices, each with different performance characteristics. Table 1 shows several heterogeneous supercomputers with more than one type of many-core devices in the TOP500 Supercomputer list (P500 Supercomputer Site, 2018). The increased diversity in the computing infrastructure leads to a true computing jungle (Seinstra et al., 2011), in which heterogeneous supercomputers with more than one type of many-core devices can be found.

In this section, we give a brief introduction to GPU Computing and explain some of the terminology used in this article. Throughout this article we will use the terminology from the CUDA programming model (Nvidia, 2018) to refer to GPU programming concepts. Although we also use other programming models in this article, CUDA is the most used programming model for GPUs.

Before we discuss our implementations, we provide a brief introduction to GPU Computing in Sec. 2. We then provide a detailed discussion of our implementation of the PRNU pattern extraction (Sec. 3), of the PRNU pattern comparison (Sec. 4), of the clustering algorithm we used for identifying common image sources (Sec. 5), and of our heterogeneous cluster application implemented using the Cashmere programming system (Sec. 6). We then evaluate the accuracy of the clustering, the performance of the Desktop application and scalability of the Cashmere application in Sec. 7 which leads us to the conclusions in Sec. 8.

### Introduction to GPU computing

In this section, we give a brief introduction to GPU Computing and explain some of the terminology used in this article. Throughout this article we will use the terminology from the CUDA programming model (Nvidia, 2018) to refer to GPU programming concepts. Although we also use other programming models in this article, CUDA is the most used programming model for GPUs.

In GPU Computing, a system consists of a host (the CPU), and one or more devices (the GPUs). The host is responsible for the memory management of the device and can move application data between host and device memory. This means that the host must allocate and free the device memory when needed. To increase the performance, data transfers between GPU memory and CPU memory may be overlapped with computation on both the GPU and the CPU.

<table>
<thead>
<tr>
<th>Name</th>
<th>Institute</th>
<th>Ranking</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jureca</td>
<td>FZJ</td>
<td>29</td>
<td>Xeon Phi 7250, K40, K80</td>
</tr>
<tr>
<td>self-made</td>
<td>Facebook</td>
<td>35</td>
<td>P100, Quadro GP100</td>
</tr>
<tr>
<td>Thunder</td>
<td>Air Force Research Laboratory</td>
<td>40</td>
<td>Xeon Phi 7120, K40</td>
</tr>
<tr>
<td>Rajjin</td>
<td>NCI-NF</td>
<td>76</td>
<td>Xeon Phi, P100</td>
</tr>
<tr>
<td>Quartetto</td>
<td>Kyushu University</td>
<td>169</td>
<td>Xeon Phi 5110P, K20, K20X</td>
</tr>
<tr>
<td>Lomonosov</td>
<td>Moscow State University</td>
<td>227</td>
<td>2070, PowerXCell Bi</td>
</tr>
<tr>
<td>Palmetto2</td>
<td>Clemson University</td>
<td>244</td>
<td>K20m, K20, K40, P100</td>
</tr>
<tr>
<td>Zeith</td>
<td>Dell HPC Innovation Lab</td>
<td>282</td>
<td>Xeon Phi 7250, Xeon Phi 7230, Xeon Phi 7210</td>
</tr>
<tr>
<td>HPCC</td>
<td>University of South Carolina</td>
<td>435</td>
<td>K20m, K40m</td>
</tr>
<tr>
<td>HC2000</td>
<td>Internet Company</td>
<td>462</td>
<td>K40, K80</td>
</tr>
</tbody>
</table>
The device is responsible for executing computational functions that are referred to as kernels. The kernels are designed to exhibit a large amount of data parallelism, allowing the same arithmetic operations to be performed simultaneously on many data elements. Kernels are executed concurrently on the device by extremely large numbers of threads. As GPU threads are much more lightweight than traditional CPU threads, kernels can be executed by millions of threads in parallel.

Threads executing the same kernel are organized in a two-level hierarchy: threads are grouped into thread blocks that are subsequently grouped into a grid. Thread blocks typically contain hundreds of threads depending on the device. Built-in variables provide threads and thread blocks with indices that direct the threads to different parts of the data. Threads in the same thread block are able to synchronize with each other, whereas there is no synchronization possible between thread blocks.

The GPU programming model supports different memory types. Global memory is a large (several GBs) high-latency off-chip memory that is typically used to store large input and output data structures. Although global memory bandwidth can be very high (up to 300 GB/s), it generally limits the performance of kernels if no other memory types are utilized. Due to the design of modern DRAMs, the peak global memory bandwidth can only be achieved if memory accesses are coalesced, that is, all threads with consecutive indices access aligned and consecutive global memory locations. Global memory is not coherent, that is, the result of a write operation is not guaranteed to be read by other threads until additional synchronization is used or the kernel finishes execution.

Shared memory is an on-chip memory that can be allocated to thread blocks and accessed at very high speed in a highly parallel manner. As all threads in a thread block can read and write to shared memory, it is an efficient way for threads to share input data and intermediate results. However, like global memory, shared memory is not coherent and as such, thread block wide synchronization is required before written values become visible to other threads.

Registers are used to store variables that are private to each thread. The number of registers used by each thread differs per kernel and is determined by the compiler. The register usage per thread block thus also depends on the number of threads in each thread block.

The performance of GPU execution often depends on how well its vast parallel resources can be utilized. The number of thread blocks that can execute in parallel on each multiprocessor within the device is bound by the dynamic partitioning of resources. Each multiprocessor has a maximum number of thread blocks and threads that it can support simultaneously. In addition, the register file and shared memory are partitioned among the concurrently executing thread blocks. The occupancy is defined as the number of threadblocks that can execute in parallel on each multiprocessor. The partitioning of multiprocessor resources often results in subtle interactions between resource limits, as a slight increase in resource usage could reduce the occupancy and result in a dramatic reduction in the achieved performance.

**PRNU pattern extraction**

Before we can compare the PRNU patterns of the images with each other, we first have to extract the PRNU pattern from the images. The extraction proceeds through a number of relatively compute-intensive steps. The number of compute operations is large, mostly due to the large number of pixels that modern digital cameras support.

In the rest of this section, we present our GPU implementation of the PRNU pattern extraction pipeline. In this and following section, we will discuss the kernels in terms of CUDA, as used the Desktop application is implemented using CUDA. As a starting point we follow, barring some modifications, the implementation for PRNU extraction as presented by Gisolf et al. (2014). Important differences with Gisolf et al. are that we leave out the digestion and quantization steps, because these steps trade-off accuracy for performance, a trade-off we do not want to make.

**Gray-scale filter**

It is well-known that processing the RGB channels individually does not result in higher accuracy when matching PRNU patterns (Chen et al., 2007; Cooper, 2013; Gisolf et al., 2014). Therefore, the RGB channels are combined into a single gray-scale channel using a straightforward GPU kernel with the standard $g = (0.299 \times r + 0.587 \times g + 0.114 \times b)$ conversion weights. The resulting gray-scale image is stored as a contiguous array of single-precision floating-point values in GPU device memory.

**Fast-noise filter**

We use the First Step Total Variation (FSTV) algorithm introduced by Gisolf et al. (2013) as the first step towards estimating the PRNU pattern. Gisolf et al. have shown that FSTV gave the best results with the lowest computation time, therefore we use this algorithm as a basis for our GPU implementations.

FSTV is denoted as:

$$W = -\nabla \cdot \left( \frac{\nabla I}{\sqrt{\|\nabla I\|^2 + \epsilon}} \right)$$  \hspace{1cm} (1)

where $W$ is the PRNU pattern, $I$ is the observed intensity for pixels of the image, $\nabla$ the gradient operator, and $\epsilon$ a small positive parameter to avoid singularities.

The gradient operator is implemented by computing the local gradient for each pixel both horizontally and vertically. The first and last pixel in a row or column are computed using forward and backward differences. The interior pixels are computed using centered differences.

Subsequently, the horizontal and vertical gradients are normalized by the magnitude of the pixel gradient. After the normalization step another gradient operator is applied to the normalized gradients in either direction.

On the GPU it is possible to combine the computation of horizontal and vertical gradients, as well as the normalization step, into a single GPU kernel. In this kernel, a single thread is created for each pixel in the image, which computes the normalized gradients for one pixel.

The final gradient operator is performed by a second GPU kernel. This separation into two steps is necessary because the last gradient operator is applied to the normalized gradients computed by the first kernel. This means that the second kernel requires data produced by different threads in the first kernel. Because there is no way to efficiently synchronize GPU device memory, the final gradient operation must be computed by a separate GPU kernel.

**Zero mean filter**

The pattern obtained by the Fast Noise filter provides a basic estimation of the PRNU pattern. However, some artifacts produced by color interpolation and on-sensor signal transfer may still be present in the pattern (Chen et al., 2008). The linear patterns produced by these artifacts could increase the correlation between images from different sensors that share the same imaging sensor design. Therefore, we apply the Zero Mean filter proposed by Chen...
et al. (2008) to ensure that the means of the even and uneven subsets of each row and each column become zero.

The Zero Mean filter is an interesting operation to implement on the GPU. The operation can be parallelized, because computing the mean of all the even (or uneven) subset of pixels in a row (or in a column) can be computed in parallel with all the other means that need to be computed. Moreover, there is even fine-grained parallelism within the computation of the means themselves. When all means have been computed, the corresponding means can be subtracted from each pixel in the image in parallel. The row-wise and column-wise operations are implemented as separate GPU kernels.

The GPU device memory is optimized for coalesced access patterns, which means that consecutive threads should access consecutive elements in memory. For the zero mean implementation this implies that computing the row-wise means by simply letting the threads iterate over the pixels in a row would result in an access pattern that gives poor performance. However, letting a number of threads equal to the width of the image iterate over the pixels in all columns in the image does result in an efficient access pattern, as shown in Fig. 1a.

It is possible to apply the zero mean filter only column-wise and use transpose operations to allow the column-wise operation to operate on what were originally the rows in the image, followed by another transpose to obtain the final result. However, implementing the zero mean filter in this way requires 4 separate GPU kernel calls that in total read all pixels 6 times and store each pixel 4 times. Transferring the entire image 10 times between the execution units on the GPU and device memory is not very efficient.

Therefore, we have implemented a GPU kernel that computes the row-wise means on non-transposed data, eliminating the need for transposes. This requires that we also parallelize the computation of the means themselves. In the row-wise GPU kernel, a single column of two-dimensional thread blocks that iterate over the pixels in a row is used, see Fig. 1b. The thread blocks iterate over the rows in steps equal to the width of the thread block, which solves the issue of coalescing because adjacent threads still access adjacent values in device memory.

After iterating over the pixels in a row, the threads that are on the same row in the two-dimensional thread block have all stored partial sums for all even and odd pixels in the row in shared memory. Then, in parallel, the threads reduce their partial sums to a single sum for the odd and even subsets of pixels in the row. Then, the same manner of iterating over the row is used to subtract the corresponding mean from all pixels in the row. For the column-wise operation, we actually use the same scheme, but for columns instead of rows, to increase the amount of parallelism in the computation.

Wiener filter

After zero mean filtering, the PRNU pattern may still contain some small-scale polygonal patterns that could be caused by imaging sensor design and JPEG compression (Chen et al., 2008). Any kind of structure in the estimated PRNU pattern that is not caused by PRNU may increase the correlation between images from different cameras that share imaging sensor design or JPEG compression. The Wiener filter is used to remove these polygonal patterns from the estimated PRNU pattern.

The Wiener filter is applied in the frequency domain. We use the CUFFT (Nvidia et al., 2018) library for the GPU implementation of Discrete Fourier Transforms (DFT). Because estimated PRNU pattern has zero mean, the computation of the minimum mean squared error reduces to the computation of the minimum squared variance.

The first step is to apply the Fourier transform to the estimated pattern and compute the square magnitudes of the frequencies. Secondly, minimum local variances are estimated using four filter sizes: \(3 \times 3\), \(5 \times 5\), \(7 \times 7\), and \(9 \times 9\). Finally, the frequencies in the estimated PRNU pattern are scaled with the ratio between the global variance and estimated minimum local variances, before applying the inverse Fourier transform.

The GPU implementation consists of multiple kernels that perform the above steps. The following kernels are relatively straightforward: computing the squared magnitudes of the frequencies, scaling the frequencies, and normalizing the pixel values.
after the inverse DFT. The kernel that estimates the local minimum variances is the most complex and time consuming.

This minimum local variance kernel is two-phased and uses two-dimensional thread blocks. In the first phase, all threads in the thread block cooperatively load the area from the input required by the thread block as a whole for computing the largest filter size. At the end of the first phase, all the input data in global memory needed by this thread block has been loaded into shared memory.

In the second phase, each thread computes, for a single element in the input, the minimum local variances for each of the four filter sizes. During the second phase, the threads reuse the sum from earlier computed smaller filters and only add the input elements on the border as filter sizes increase; this eliminates the otherwise redundant computations.

The GPU kernel that computes the global variance of the estimated PRNU pattern is a simple reduce operation that squares and sums the input values. It is executed twice: in the first iteration each thread block produces a per thread block sum squared value, while in the second iteration a single thread block is used to reduce the output of the first iteration to a single value.

After applying this Wiener filter, the estimated PRNU patterns of the images are ready to be compared.

Pattern comparison

It is our goal to cluster images based on the extracted PRNU patterns. However, this requires comparing the PRNU patterns of all images with each other, resulting in $\mathcal{O}(N^2)$ comparisons. As such, comparing the images is the most expensive part of the application.

Peak to Correlation Energy (PCE) ratio is a method for comparing PRNU patterns (Chuang et al., 2011; Fridrich, 2009; Bayram et al., 2012). Goljan et al. (Goljan, 2008) have shown that PCE is a much more suitable detection statistic, compared to Normalized Cross Correlation (NCC) for the related problem of camera identification. This is because periodic signals, e.g. linear patterns, in both PRNU patterns will increase their correlation, while reducing the PCE score.

However, compared to NCC, performing the comparisons with PCE increases the complexity of the most expensive part of the computation as PCE is computed in the frequency domain. For larger images and images whose dimensions are not a simple composite of powers of two, the Fourier transforms are the most computationally expensive part of the comparison.

An additional complexity is that the PRNU patterns are quite large and it is not uncommon that hundreds of patterns need to be compared. In such cases, the patterns will not fit in the GPU device memory or even in the main memory of the host.

For this purpose, we have implemented a software cache that allocates as much memory as possible to store PRNU patterns in main memory. Patterns that have not been used for a while are evicted from the cache, according to a least-recently used (LRU) cache policy. However, instead of storing the patterns to disk, the cache simply forgets and recomputes the pattern on the fly when needed. This is because reading the JPEG image and recomputing the pattern on the GPU is faster than storing and loading the pattern to and from disk.

PCE is computed as the ratio between the height of the peak and the energy of the cross correlation between two PRNU patterns. To compute the PCE score of two PRNU patterns $X$ and $Y$, the following steps are required:

1. Patterns $X$ and $Y$ need to be present in GPU device memory.
2. A GPU kernel converts $X$ and $Y$ from single-valued floating-point arrays into zero-padded complex-valued arrays. Pattern $Y$ is also flipped both vertically and horizontally.

3. The CUFFT library is called to transform both patterns into frequency domain.
4. A GPU kernel is used to compute the cross correlation.
5. The CUFFT library is used to perform the inverse transform on the cross correlation.
6. GPU kernels are used to find the peak and compute the energy of the cross correlated signal.

Taking the above into consideration, we have the following approach: In GPU memory, all of the available space is reserved for storing as many patterns as possible. The host program steers the computation in a way that minimizes both the amount of data transferred between host and device memory, as well as the number of forward Fourier transforms that need to be applied. The GPU kernels used in step 2 are straightforward as they implement a simple copy operation that reads the pattern and stores it as the real component of a complex-valued array, optionally also flipping the pattern horizontally and vertically.

The GPU kernel in step 4 computes the cross correlation of a flipped and non-flipped pattern that have been transformed to frequency domain. In theory, this kernel could be implemented such that it computes multiple cross correlations at once reusing some of the pattern data on chip. However, computing multiple cross correlations at once would also require storing multiple cross correlations in GPU device memory, which in turn would reduce the number of Fourier transformed patterns that we can cache in GPU memory. As such, computing multiple cross correlations with a single GPU kernel would increase the number of transfers between host and device and the number of forward Fourier transforms, and therefore decrease overall application performance.

The GPU kernels in step 6 implement reduction operations that reduce the cross correlated signal to three values: the peak, the location of the peak, and the energy of the signal. The energy is computed as the average of the correlated signal minus an $11 \times 11$ square area around the peak. Only the value of the peak and the energy are transferred back from GPU memory to the host, where these two values are divided to obtain the PCE score that expresses the similarity between the two compared patterns.

Clustering algorithm

This section presents the clustering algorithm that we use to cluster the images based on the computed all-to-all PCE scores. It is important to understand that the clusters formed in this step express the decision of the algorithm on whether or not two images originated from the same source. In other words, if two images end up in the same cluster, we conclude that those two images were created using the same imaging sensor.

Before presenting our implementation, we provide a quick overview of related work in clustering PRNU patterns. Bloy (2008) appears to be the first to investigate clustering images based on camera fingerprints without the presence of subject cameras. For clustering, they use only the $1024 \times 1024$ center region of the green color plane, to reduce computations and to avoid the need for filtering out color interpolation effects as discussed in Section 3.3. Their clustering algorithm takes a somewhat hierarchical approach, but instead of averaging the similarities they average the patterns and recompute the similarities. Their goal is to establish an estimation of the camera fingerprint, allowing at most 50 patterns to be averaged to reconstruct the fingerprint. The other images are compared against the estimated camera fingerprints, and if their similarity score exceeds a selected threshold, the image is added to the cluster.

Caldelli et al. (2010) and Gisolf et al. (2014) use a hierarchical clustering algorithm with the average linkage method, where inter-
cluster similarities are computed as the average of the similarities between the images in the clusters. Both papers use NCC as their similarity metric. Caldelli et al. use the silhouette coefficient to determine when to stop clustering. Gisolf et al. require the user to provide a data set dependent threshold.

Li (2010) claims that given the large number of images, analyzing the images in full size is computationally infeasible. As such, Li uses NCC and only considers a small part of the images to further reduce computational complexity. As for their clustering algorithm, they take a subset of the data as a training set and iteratively use k-means clustering to train a classifier with which the rest of the data is classified. In this article, we demonstrate that such approximations are not necessary as by using GPUs we can make analyzing the full images and perform an all-to-all comparison computationally feasible.

Our goal is to provide a GPU solution to the problem of clustering PRNU patterns. As such, we use a hierarchical clustering approach with averaged inter-cluster distances, which seems to be the most common approach in the literature. However, the main difference with related work in clustering PRNU patterns is the fact that we use PCE as a similarity metric rather than NCC.

The current state-of-the-art in hierarchical clustering is the fastcluster package by M"ullner (M"ullner, 1105), which is also used in SciPy (SciPy Reference Guide and ht, 2018) and R (fastcluster, 2018). However, most existing software packages work with dissimilarities rather than the similarity scores that are produced by PCE. We could convert the PCE scores to dissimilarities to work with existing clustering packages. However, we do not want to distort the distribution of scores in any way, because we aim to use a well-known threshold for image source identification using PCE scores. Goljan and Fridrich (2013) have found that a PCE score of 60 corresponds to a probability of 10^{-5} of falsely identifying an image as taken by a specific camera. We aim to use this threshold to objectively decide whether two images originate from the same source. Therefore, we implemented our own hierarchical clustering algorithm to work with similarity scores.

**Cashmere implementation**

In contrast to the Desktop application (Software repository of thb; van Werkhoven and Hijma, 2018) that supports one NVIDIA GPU, the Cashmere implementation (Software repository of the; Hijma and Jacobs, 2018a) is aimed at scaling up the computation with all kinds of many-core devices available in a compute-cluster, something we call the compute jungle (Seinstra et al., 2011). Examples are different GPUs from NVIDIA and AMD, all with their own performance characteristics. These different performance characteristics make it difficult to obtain good speed-up, since if each many-core device gets an equal share of the work, some devices would be finished earlier than others. To compensate for this problem, we make use of Cashmere (Hijma et al., 2015a; Software repository of thb; Hijma and Jacobs, 2018c).

Cashmere is a framework that seamlessly glues together two other systems, namely MCL and Constellation. The MCL framework (Hijma et al., 2015b; Software repository of thc; Hijma and Jacobs, 2018b) allows one to write kernels for many-core devices. A difference with CUDA is that one can write kernels on multiple levels of abstraction. On a high level, the kernels are relatively easy to write, portable to different kinds of many-core devices, but the performance may lack. The lower the level of abstraction is, the more room for optimization is exposed to the programmer, but portability may lack. With MCL, programmers start at a high level and then move to lower levels when needed using guidance from a compiler.

Constellation (Maassen et al., 2011; Software repository of thd; Jacobs et al., 2018) is a framework that allows one to write an application for a compute cluster in terms of activities. Constellation’s task is to schedule these activities efficiently among the compute nodes of the cluster and it achieves this by work-stealing: if a compute node runs out of work, activities will be stolen from other compute nodes, thus balancing the work-load evenly among the compute nodes, which is essential to achieve good scalability. Constellation is unique in the flexibility it provides to map activities to compute nodes. In particular it can handle highly heterogeneous resources using **context-aware work stealing** that automatically matches activities to suitable nodes in the compute cluster.

Cashmere connects Constellation and MCL, makes it convenient to call MCL kernels, and is responsible for overlapping data transfers to and from the device with kernel execution. In essence, multiple activities can submit kernel executions to the many-core device and Cashmere ensures that this is done efficiently.

Since the Cashmere application targets compute jungles, it has to handle more types of parallelism than the Desktop application. First, there are multiple CPU threads within a compute node that submit kernels, and second, there are multiple compute nodes within the cluster.

This has consequences for the application: The kernels used in the Cashmere application are highly similar to the CUDA kernels but written in MCL. We have chosen to write kernels with high portability instead of maximum optimization. In addition, since the MCL compiler produces OpenCL code for portability, the CUFFT library has been replaced with the OpenCL clFFT library (Advanced Micro Devices and c, 2018). This library is not as fast as CUFFT (sometimes, even a factor two slower), but it is portable to different many-core devices. The largest consequence is for the overall design as discussed below.

First of all, we have to subdivide the work as evenly as possible over various compute nodes. We have to compare N images with N − 1 images which results in a correlation matrix of N × N. The correlation matrix is symmetric, so we are only interested in the area below (or above) the diagonal. Fig. 2 shows an example correlation matrix of 100 images subdivided over four nodes A, B, C, and D. Suppose that we assign to node A images 0–25, images 25–50 to node B, etc. This assignment maps the four triangular parts to each node, as depicted in Fig. 2. We will call these parts **triangular jobs**. Then there are six square areas — we will call them **square jobs** — that will contain correlations with images that can be assigned to two nodes. For example, the square job (0–25,25–50) can be assigned to node A (since this node has images 0–25 and D. Suppose that we assign to node A images 0–25, images 25–50 to node B, etc. This assignment maps the four triangular parts to each node, as depicted in Fig. 2. We will call these parts **triangular jobs**. Then there are six square areas — we will call them **square jobs** — that will contain correlations with images that can be assigned to two nodes. For example, the square job (0–25,25–50) can be assigned to node A (since this node has images 0–25

![Fig. 3. Subdivision of the square job (0–25,25–50) assigned to node A in two levels of 4 squares. An example is (0–12,25–37), that itself is subdivided into (0–7,25–32), etc.](image)
to cluster images of the same resolution, see Table 2, named after parameter computer 5 (DAS-5) (Bal et al., 2016), specifically the compute cluster equipped with (possibly different kinds of) GPUs. Furthermore, forensic institutions or large police departments that have access to supercomputers with a powerful GPU. The Cashmere application targets a scenario is a crime laboratory with a powerful Desktop PC equipped with both applications presented in this article. The Desktop application focuses on moderately large data sets and a use-case of both applications. To measure our performance, the GTX Titan X GPU has 3072 CUDA cores running at 1.08 GHz and 12 GB of device memory.

We create an activity for each of those square jobs and assign it as evenly as possible to the compute nodes. This means that in principle this node will execute these correlations, but other nodes may steal these correlations when they run out of work. Since nodes A and B have twice the amount of work of node C and D in the subdivision of Fig. 2, it is likely that node C and D will start stealing work from nodes A and B at some point. The Constellation run-time handles this automatically and efficiently. Cashmere uses multiple CPU threads to issue work to the GPU. Therefore, if one thread runs out of work, the other threads keep the GPU busy while work is being stolen.

As mentioned in Sec. 4, the correlations are maximally efficient if all the noise patterns reside in the memory of the device. Since a square or triangular job in Fig. 3 may represent a number for noise patterns larger than what fits in memory, these jobs may be subdivided in a quad tree-like manner as depicted in Fig. 3 for a square job. A job is hierarchically subdivided in new jobs of which each can be subdivided into four additional ones and so forth. This scheme has the additional benefit that besides large jobs, small sub jobs may also be stolen which helps balancing the load towards the end of the execution.

Some data may overlap in the noise patterns represented by the different jobs in successive or parallel executions of these jobs – for example the square jobs (0–7,25–32) and (0–7,32–39) have reuse for patterns 0 to 7 – the cache in the Cashmere application works a bit differently than the Desktop application cache. The difference is that the Cashmere application has a two-level cache, a small cache on the device, and a large cache in the memory of the compute node. The cache is also capable of handling multiple CPU threads at the same time.

Another difference is that the noise patterns are computed on-demand and stored in the memory cache and the device cache. When two CPU threads require the same noise patterns, they automatically cooperate to fill the cache, such that both CPU threads can benefit as to execute the correlations as fast as possible.

When all noise patterns of a job are on the device, all the PCE correlations are computed at once. The resulting correlations will be sent to the parent activity. For example, an activity responsible for (0–7,25–32) sends the correlations to the parent activity that is responsible for (0–12,25–39). As soon as this activity receives all its correlations, it will pass them on to its parent (0–25,25–50) and so on until all activities reach the activity that is responsible for the whole correlation matrix.

**Evaluation**

This section presents the performance and qualitative evaluation of both applications presented in this article. The Desktop application focuses on moderately large data sets and a use-case scenario is a crime laboratory with a powerful Desktop PC equipped with a powerful GPU. The Cashmere application targets forensic institutions or large police departments that have access to a compute cluster equipped with (possibly different kinds of) GPUs.

The hardware platform that we use is the Distributed ASCI Supercomputer 5 (DAS-5) (Bal et al., 2016), specifically the compute cluster at the VU University Amsterdam. The cluster consists of 68 compute nodes with two 8-core 2.4 GHz Intel Haswell E5-2630-v3 CPUs and 64 GB of main memory. Of these compute nodes, 16 nodes are equipped with NVIDIA GTX Titan X Maxwell GPUs which we mainly use to measure our performance. The GTX Titan X GPU has 3072 CUDA cores running at 1.08 GHz and 12 GB of device memory.

The data set that we use is part of the Dresden image database (Gloe and Böhme, 2010). Specifically we use three subsets to cluster images of the same resolution, see Table 2, named after one of the camera brands that were used to create the images. The Pentax and Praktica data sets are moderately large and their noise patterns fit in the main memory of a single DAS-5 node. The Olympus data set is very large and its noise patterns will not fit in the main memory of a single DAS-5 node.

Before discussing the quality of the clusterings produced by our applications, it is important to understand that we have not filtered the data set in any way. It is difficult to compare our clustering results to related work, since many filter out images that were over or under exposed (Lin and Li), or construct their own image database from not publicly available data sets (Gisolf et al., 2014; Caldelli et al., 2010; Fahmy, 2015; Li, 2010). We want to ensure that our results can easily be verified and reproduced, therefore we only use publicly available data without any form of (random) selection or filtering. Another note we want to make is that the Dresden image set is made specifically to make it challenging to cluster these images based on the camera.

**Quality of clustering**

We have performed our quality analysis using the Pentax subset. The results that we obtain here are representative for the other subsets.

Hierarchical or agglomerative clustering starts from the set of singletons, clusters of size one, that will be grouped into larger clusters. The main output data structure is the linkage, which can be seen as an ordered list of the clusters merged during clustering. The resulting linkage is often visualized using a dendrogram, which is a tree like representation, used to provide insight in the order and the similarity or dissimilarity of the clusters merged at each step in the algorithm.

Fig. 4 shows such a dendrogram together with the similarity matrix for the Pentax data set. Note that the similarity matrix is

---

**Table 2**

<table>
<thead>
<tr>
<th>name data set</th>
<th>Pentax</th>
<th>Praktica</th>
<th>Olympus</th>
</tr>
</thead>
<tbody>
<tr>
<td>image dimensions</td>
<td>3000 x 4000</td>
<td>1920 x 2560</td>
<td>2736 x 3648</td>
</tr>
<tr>
<td>number of images</td>
<td>638</td>
<td>1095</td>
<td>4980</td>
</tr>
<tr>
<td>fits in main memory</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

---

**Fig. 4.** Dendrogram and similarity matrix for the Pentax data set. In the dendrogram we used the same color for relations with a PCE score higher than 60.0. In the similarity metric, each pixel represents a pair of images compared, its color represents the PCE score. (For interpretation of the references to color/colour in this figure legend, the reader is referred to the Web version of this article.)
sorted using the same order as shown in the dendrogram. Because the matrix is sorted, the four clusters in the data set can also be seen in the similarity matrix as four square areas in which most images have a high similarity to each other.

It is possible to construct a flat clustering from the hierarchical clustering. Essentially, this means cutting the dendrogram at a certain height. To construct a flat clustering, the algorithm takes the earlier computed linkage and only considers the merges of clusters whose inter-cluster similarity is above a threshold. The threshold used to color the dendrogram in Fig. 4 is set at 60.0, which according to Goljan and Fridrich (2013) corresponds to a probability of 10^{-5} of falsely identifying an image as taken by a specific camera.

We interpret the flat clustering as a classification of the relations between all images in the data set. The relation between two images is negative when they originate from different cameras and positive when they originate from the same camera. However, if the images in a balanced data set originate from four different cameras, then the negatives will be four times as prevalent as positives. Therefore, we use positive predictive value (PPV) and negative predictive value (NPV) as these metrics are robust against the prevalence of one class over the other. Fig. 5 shows the PPV and NPV for different thresholds of the Pentax data set, clustered with hierarchical clustering using averaged inter-cluster similarities based on PCE scores.

For thresholds 0 and 1 all images are in a single cluster, therefore there are no negative relations and as such the NPV is 1. The PPV is only 0.25 because all relations are classified as positive, while only 25% of the relations are actually positive in the ground truth. At threshold 60, we can see that the PPV is 0.99996, which means that there is exactly one false positive. However, the NPV is 0.86, which means that only 86% of the relations identified as negative are truly negative. Actually, as of thresholds larger than 37 the PPV remains

Fig. 5. The positive and negative predictive values for different thresholds of the Pentax data set, clustered with hierarchical clustering based on PCE similarity scores.

Fig. 6. The images and their post-processed PRNU patterns that caused the only false positive in the Pentax data set at thresholds > 37. The PRNU patterns have been post processed to emphasize the image structure in the pattern. These are images “Pentax_OptioA40_1_31217” and “Pentax_OptioA40_3_32419” from the Dresden image database (Gloe and Böhme, 2010).
constant at 0.99996, meaning that there is exactly one pair out of the 638 images that is clustered together while they were created with different cameras.

Fig. 6 shows this particular pair of images and their corresponding PRNU patterns. Note that the PRNU patterns have been post processed aggressively to highlight all the structure of the image that leaked into the pattern. This post processing is necessary as PRNU patterns otherwise appear as almost entirely flat grayscale fields. We can see that after post processing, quite some structure in the image can be reconstructed from the PRNU pattern. We believe that the common structure in these two images caused a high PCE score for this particular pair. This is of course not desirable, but cannot always be avoided due to how patterns are extracted from images.

Performance of the GPU desktop application

In this section, we discuss the performance of our GPU implementation (Software repository of van Werkhoven and Hijma,

<table>
<thead>
<tr>
<th>name data set</th>
<th>Pentax</th>
<th>Praktica</th>
</tr>
</thead>
<tbody>
<tr>
<td>populating the cache</td>
<td>120.7s</td>
<td>122.1s</td>
</tr>
<tr>
<td>computing similarity scores</td>
<td>20 m 45s</td>
<td>24 m 33s</td>
</tr>
<tr>
<td>computing linkage</td>
<td>0.2802s</td>
<td>0.7866s</td>
</tr>
<tr>
<td>total</td>
<td>22 m 46s</td>
<td>26 m 36s</td>
</tr>
</tbody>
</table>

Fig. 7. A breakdown of the execution times of different parts of clustering. Note that the computing linkage step is too small to be visible in the graph.

(a) On an NVIDIA GTX Titan X GPU, CUDA and CUFFT version 8.0 (measured in ms).

(b) On the CPU, OpenJDK 1.8.0, JTransforms 2.4 (measured in s).

Fig. 8. Timelines for PRNU Pattern extraction of a 3000 x 4000 image from the Dresden image database.
2018) for extracting and comparing PRNU patterns. We will compare the execution times of the CPU and GPU version, but only to give an intuition about the performance improvement as a full architectural comparison of GPU versus CPU is outside the scope of this article. Please note, that this is a comparison between different applications, rather than a comparison of which architecture is best suited for the application. In addition to the architecture many other factors influence the performance: first of all, the Java application is not optimized for performance, both applications use different FFT libraries, programming languages, compilers, and so on. However, the increased performance provided by our GPU-enabled application does allow us to study the problem of clustering images based on their PRNU patterns on a much larger scale and with greater accuracy.

We evaluate the Desktop application with the two moderately large data sets Pentax and Praktica (see Table 2) that each completely fit in the memory of a reasonably powerful Desktop computer. Table 3 shows the precise execution times for these two data sets. Fig. 7 shows the time visually where it is clear that computing similarity scores is of $O(N^2)$, whereas extracting noise patterns to populate the cache is $O(N)$.

To give an intuition about how we obtain these high performance numbers, in Fig. 8, we show two Gantt charts of the PRNU pattern extraction for a 3000 $\times$ 4000 image from the Dresden image database for the GPU and CPU version. The gray bar shows the duration of the filter that converts the image into gray-scale. The blue bars represent the normalized gradient and gradient operators in the Fast Noise filter. The horizontal and vertical filtering as part of the Zero Mean filter are colored green. The Wiener filter is both red and yellow, to distinguish between GPU kernels we have implemented and the kernels used within the CUFFT library (similarly for the CPU version a Java FFT library). Table 4 shows the precise execution time for the kernels. To give an indication of the performance improvement experienced by the user, extracting the same image using our single-threaded Java application took 3.15 s, as opposed to 15.8 ms using our GPU application written in Java/ CUDA.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Execution times in milliseconds per kernel for the PRNU pattern extraction, kernels from the CUFFT library have been summarized into Forward FFT and Inverse FFT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>grayscale</td>
<td>0.365</td>
</tr>
<tr>
<td>normalized_gradient</td>
<td>0.752</td>
</tr>
<tr>
<td>gradient</td>
<td>0.637</td>
</tr>
<tr>
<td>computeMeanVertically</td>
<td>0.641</td>
</tr>
<tr>
<td>computeMeanHorizontally</td>
<td>0.652</td>
</tr>
<tr>
<td>toComplex</td>
<td>0.568</td>
</tr>
<tr>
<td>Forward FFT</td>
<td>4.018</td>
</tr>
<tr>
<td>computeSquaredMagnitudes</td>
<td>0.535</td>
</tr>
<tr>
<td>computeVarianceEstimates</td>
<td>1.959</td>
</tr>
<tr>
<td>computeVarianceZeroMean</td>
<td>0.198</td>
</tr>
<tr>
<td>sumFloats</td>
<td>0.004</td>
</tr>
<tr>
<td>scaleWithVariances</td>
<td>0.914</td>
</tr>
<tr>
<td>Inverse FFT</td>
<td>3.995</td>
</tr>
<tr>
<td>normalizeToReal</td>
<td>0.534</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>15.77</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Execution times in milliseconds per kernel for PCE comparison, kernels from the CUFFT library have been summarized into Forward FFT and Inverse FFT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>toComplexAndFlip</td>
<td>0.600</td>
</tr>
<tr>
<td>Forward FFT</td>
<td>4.140</td>
</tr>
<tr>
<td>toComplex</td>
<td>0.575</td>
</tr>
<tr>
<td>Forward FFT</td>
<td>3.939</td>
</tr>
<tr>
<td>computeCrossCorr</td>
<td>1.084</td>
</tr>
<tr>
<td>Inverse FFT</td>
<td>3.933</td>
</tr>
<tr>
<td>findPeak</td>
<td>0.350</td>
</tr>
<tr>
<td>maxloc_Floats</td>
<td>0.006</td>
</tr>
<tr>
<td>computeEnergy</td>
<td>0.389</td>
</tr>
<tr>
<td>sumDoubles</td>
<td>0.005</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>15.02</strong></td>
</tr>
</tbody>
</table>

(a) On an NVIDIA GTX Titan X GPU, CUDA and CUFFT version 8.0 (measured in ms).

(b) On the CPU (measured in ms).

Fig. 9. Timelines for a PCE comparison of two 3000 $\times$ 4000 image from the Dresden image database.
Fig. 9 shows the Gantt charts of the comparison with PCE of two 3000 × 4000 images. Again, we make a distinction between the kernels that we have implemented and the kernels used within the CUFFT library (or the Java FFT library). The first two yellow kernels and forward FFTs bring two noise patterns in the frequency domain (the first kernel also flips the PRNU pattern horizontally and vertically). The blue bar represents the computation of the cross correlation after which the inverse FFT is executed. Finally, two reductions are performed to find the peak and compute the energy. Table 5 shows the precise execution times. All these performance numbers have been obtained with the NVIDIA Profiler. We see here that the CPU version takes 558 ms, while our version takes 15.0 ms. Again, we do not want to compare our GPU version with the CPU version, but our main point is that our GPU are able to extract and compare images with a very high rate, making it possible to process large data sets without reducing accuracy.

## Performance of the Cashmere application

The single-node performance of the Cashmere application (Software repository of the; Hijma and Jacobs, 2018a) is in general lower than the Desktop application (Software repository of the; van Werkhoven and Hijma, 2018). This is due to the fact that this application targets different types of GPUs and uses a more generic FFT library that supports both NVIDIA and AMD GPUs and ordinary CPUs, whereas CUFFT is highly optimized for NVIDIA GPUs.

However, instead of absolute performance, this application aims at high scalability. The main goal is that adding resources means higher performance even in a compute jungle where many resources may have different performance characteristics, which usually makes scalability a problem.

Before we look into the scalability, we compare the performance

<table>
<thead>
<tr>
<th>name data set</th>
<th>Pentax</th>
<th>Praktica</th>
<th>Olympus</th>
</tr>
</thead>
<tbody>
<tr>
<td>image dimensions</td>
<td>3000 × 4000</td>
<td>1920 × 2560</td>
<td>2736 × 3648</td>
</tr>
<tr>
<td>number of images</td>
<td>638</td>
<td>1095</td>
<td>4980</td>
</tr>
<tr>
<td>fits in main memory</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>time Desktop</td>
<td>22 m 46s</td>
<td>26 m 36s</td>
<td>43 h 33 m</td>
</tr>
<tr>
<td>time Cashmere (1 node)</td>
<td>47 m 14s</td>
<td>44 m 44s</td>
<td>53 h 25 m</td>
</tr>
</tbody>
</table>

![Fig. 10. The relative execution time of the Cashmere application (single node) compared to the Desktop application.](image)

![Fig. 11. Speedup of the Cashmere application compared to perfect speedup on the three data sets.](image)
of the Desktop application with the Cashmere application executing on a single node. Table 6 shows the execution times for the three data sets for the Desktop and the Cashmere application on a single node. Fig. 10 shows the relative execution times of the Desktop application (with value 1) compared to the Cashmere application. We can see that the Cashmere application is in general slower, ranging from 110% to little over a factor 2. The difference is mostly the FFT calculation that dominates the execution. The more powers 2 in the dimensions of the images, the more the performance differs between the two FFT libraries, because factors of two in the sizes of FFTs allow many hardware-specific optimization opportunities that the CUDA FFT library is able to exploit.

For the scalability studies, we compare the execution times of the Cashmere application with different numbers of a single GPU type, in this case the Titan X. Fig. 11 shows the scalability up to 16 Titan X GPUs for the three data sets compared to ideal speedup. We measure strong scalability which means that the total amount of work remains the same for each configuration. Strong scalability is in general more difficult to achieve than weak scalability where the amount of work per node remains the same irrespective the number of nodes.

In Fig. 11 we see that the Pentax set scales linearly, the Olympus set scales superlinearly, and the Praktica set scales somewhat worse than the other two data sets. The cause for this lower scalability is in essence that this data set is too small for this number of nodes. Table 7 shows the execution times of the three data sets when run with 16 nodes with a Titan X GPU. The table also shows the image size, number of images, the number of images that fit in the device cache, and the number of jobs that are submitted to the GPU (these jobs represent the smallest jobs in the quad tree representation of Fig. 3). Although the Pentax and Praktica data set are comparable in size with respect to execution time, the Praktica set scales worse because there are not enough jobs for the nodes to properly do load balancing, which means that some nodes run out of work while there are no more jobs available. The number of jobs depends on how many images fit in the device cache: the more in the cache, the larger and fewer the jobs. The number of images in the cache is inversely related to the image size, which is smaller in the Praktica set than in the Pentax set.

Although Cashmere does not scale optimally on the small Praktica data set, Cashmere shows excellent scalability for the Olympus set, the kind of sets the Cashmere application targets. The superlinear speedup for the Olympus set can be explained by the fact that we have 16 nodes with also 16 times the memory. The Olympus set does not fit into the memory of 1 node, which means that PRNU patterns need to be recomputed. However, the work is subdivided among 16 nodes, which means that the images a node works on will generally fit in memory, limiting the number of times that nodes have to recompute PRNU patterns.

The Cashmere application also performs excellently with different types of GPUs. Table 8 shows the execution times of the Olympus set with different hardware configurations. The third row shows the performance with all the available GPUs in the DAS-5. All the GPUs have 12 GB of memory except the K20 that has 5 GB. The K20 is also considerably slower than the other cards, followed by the K40, the Titan X, and the TitanX-Pascal as the fastest. To show how well Cashmere is able to balance the load, we show a Gantt chart of the last 10 min of the execution with 4 Titan X, the K20, Providing Table 7

Table 7

<table>
<thead>
<tr>
<th>name data set</th>
<th>Pentax</th>
<th>Praktica</th>
<th>Olympus</th>
</tr>
</thead>
<tbody>
<tr>
<td>image dimensions</td>
<td>3000 × 4000</td>
<td>1920 × 2560</td>
<td>2736 × 3648</td>
</tr>
<tr>
<td>number of images</td>
<td>638</td>
<td>1095</td>
<td>4980</td>
</tr>
<tr>
<td>#images in device cache</td>
<td>32</td>
<td>136</td>
<td>63</td>
</tr>
<tr>
<td>#jobs</td>
<td>2075</td>
<td>1128</td>
<td>73920</td>
</tr>
<tr>
<td>time 1 node</td>
<td>47 m 14s</td>
<td>44 m 44s</td>
<td>53 h 25 m</td>
</tr>
<tr>
<td>time 16 nodes</td>
<td>2 m 55s</td>
<td>3 m 16s</td>
<td>3 h 10 m</td>
</tr>
<tr>
<td>speedup</td>
<td>16.1</td>
<td>13.7</td>
<td>16.9</td>
</tr>
</tbody>
</table>

Table 8

<table>
<thead>
<tr>
<th>configuration</th>
<th>#nodes</th>
<th>execution times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Titan X</td>
<td>1</td>
<td>53 h 25 m</td>
</tr>
<tr>
<td>16 Titan X</td>
<td>16</td>
<td>3 h 10 m</td>
</tr>
<tr>
<td>16 Titan X, 1 K20, 1 K40, 2 TitanX-Pascal</td>
<td>20</td>
<td>2 h 44 m</td>
</tr>
<tr>
<td>8 Titan X</td>
<td>8</td>
<td>6 h 27 m</td>
</tr>
<tr>
<td>4 Titan X, 1 K20, 1 K40, 2 TitanX-Pascal</td>
<td>8</td>
<td>7 h 05 m</td>
</tr>
</tbody>
</table>

Fig. 12. A Gantt chart of the last 10 min of the execution of configuration 4 Titan X, 1 K20, 1 K40, and 2 TitanX-Pascal on the Olympus data set. Each machine has 4 CPU threads issuing jobs to the GPU. Each pink block is the smallest job in the quad tree representation as explained in Fig. 3. (For interpretation of the references to color/colour in this figure legend, the reader is referred to the Web version of this article.)
K40, and 2 TitanX-Pascal GPUs in Fig. 12, where each machine has 4 CPU threads issuing jobs to the GPU. We can see that each thread finishes roughly at the same time, which shows that the work has been divided well.

Other things to notice in Fig. 12 are that the K20 submits significantly more jobs than the other nodes. Since the K20 has only 5 GB of memory, the jobs are smaller than for the other GPUs as not as much PRNU noise patterns simultaneously fit in the memory. We also notice that the K40 is slower than the other ones and the TitanX-Pascals are slightly faster than the Titan X’s (about 10% faster when counting the jobs). Finally, the durations of some jobs are shorter than others within the same GPU. This can be explained by multiple CPU threads issuing jobs resulting in some CPU threads having to wait or by a compute node executing a stolen job that is smaller than the other jobs.

Discussion

In this section we have shown excellent results in clustering images based on the PRNU pattern in three areas:

- **High accuracy of clustering images** We have used the best current techniques to increase the quality of clustering.
- **High performance for small to medium data sets** We have shown a Desktop application that has very high performance and allows an investigator to get results in a reasonable time for small to medium data sets.
- **Excellent scalability for large data sets** We have shown the Cashmere application that can make the most out of a cluster computer with GPUs. In addition, for this application one can add any type of GPU to increase the speed of the application.

Our choice was to get high quality results, for example by using the more expensive PCE instead of NCC, leaving out digitization, quantization, and cropping to show what is possible when making the most of the available hardware. However, to further increase the sizes of the data sets, it is possible to add those steps to be able to scale up to even larger data sets.

Conclusions

Analyzing common image sources is of high importance to digital forensic investigations due to the prevalence of images. This article focused on an important but highly compute-intensive study. In: Electronic Imaging 2007. International Society for Optics and Photonics, 65050P–65050P.


Lin, C.-T. Li, Large-scale image clustering based on camera fingerprints, IEEE Transactions on Information Forensics and Security.


