Large-scale Fingerprint Identification on GPU

Raffaele Cappelli, Matteo Ferrara, and Davide Maltoni

Abstract—This paper proposes a new parallel algorithm to speed up fingerprint identification using GPUs. A careful design of the algorithm and data structures, guided by well-defined optimization goals, yields a speed-up of 1946x over a baseline sequential CPU implementation and of 207x over a CPU implementation optimized with SIMD instructions. The proposed algorithm enables a medium-scale AFIS (Automated Fingerprint Identification System) to run on a simple PC with four Tesla C2075 GPUs. On a benchmark with 250000 fingerprints and 100000 queries, the proposed system yields state-of-the-art biometric accuracy with a throughput of more than 35 million fingerprint matches per second. The proposed approach can be easily scaled-up, thus making possible the implementation of a large-scale AFIS (i.e., with a database of hundred million fingerprints) on inexpensive hardware.

Keywords: Fingerprint identification, minutiae, GPU, CUDA, SIMD, MCC.

1. Introduction

Biometric systems are widely used both in forensic and civil applications to automatically recognize the identity of a person on the basis of physiological or behavioral characteristics [24]. Examples of biometric applications span from physical access control to forensic identification, from border crossing to voters authentication. Although many biometric traits (e.g., fingerprints, face, iris, voice, etc.) have been thoroughly studied, fingerprints, due to their peculiarities (i.e., individuality, persistence, cost and maturity of the products), remains the most used one [37].

1.1. Fingerprint matching and identification

A *fingerprint* is the representation of the epidermis of a finger: it consists of a pattern of interleaved ridges and valleys (Fig. 1.a). Discontinuities in the ridges (e.g., terminations or bifurcations) are called *minutiae* (Fig. 1.a): nowadays, most state-of-the-art fingerprint recognition algorithms are based on minutiae matching [BO]. Each minutia can be described by some attributes, including its location in the fingerprint, its direction, its type (termination or bifurcation), and a value representing the quality of the fingerprint pattern in its neighborhood. Minutia-based matching algorithms usually consider each minutia as a triplet $m = \{x, y, \theta\}$ encoding the spatial coordinates and the ridge angle. The set of minutiae attributes extracted from a fingerprint is called *template*; fingerprint matching consists in comparing two templates to determine whether the two sets of minutiae come from the same finger. Two minutiae are considered matching if the spatial distance between them is smaller than a given spatial threshold and their directional difference is smaller than a given angular threshold; such thresholds are necessary to compensate for the unavoidable errors made by minutiae extraction algorithms and to account for small elastic distortions due to skin elasticity. In practice, minutiae matching is an "extended" point pattern matching problem. Unfortunately, neither the alignment parameters nor the point-correspondence function are known a priori, hence solving the matching problem is hard. A brute force approach, that is evaluating all the possible solutions, is exponential in the number of minutiae, therefore suboptimal heuristics are typically used in real applications. The first automatic algorithms, developed in the early 50s, were inspired by the manual techniques of forensic experts and were aimed at determining the global (rigid) alignment leading to an optimal spatial (and directional) minutiae pairing: Hough transform was a common solution [47]. In the last decades, with the progress in the field of computational intelligence, many other techniques were proposed to improve fingerprint matching [25], including machine-learning methods [33], evolutionary algorithms [49] [50], and fuzzy similarity measures [16]. More recently, researchers have focused on local matching algorithms, based on fixed-length features that characterize the neighborhood of each minutia [4] [10] [53] [40]; these features are usually represented as bit-vectors, allowing efficient similarity measures to be implemented on many hardware architectures and simplifying the design of template protection methods [53] [19].

Although state-of-the-art approaches are nowadays very accurate and able to tolerate common perturbations (translation, rotation, deformation, missing or spurious minutiae, etc.) [37], the computational-demanding nature of minutiae-based matching still makes the development of large-scale fingerprint identification systems challenging in terms of efficiency. In fact, the time required to search a query fingerprint on a database grows with the size of the database itself: when the database contains millions of fingerprints, very expensive hardware platforms are required to operate at high throughput. This is the case of the AFIS owned by police agencies such as FBI, or the huge civil identification systems being deployed

The authors are with the Department of Computer Science and Engineering of the University of Bologna, 47521 Cesena (FC), Italy (e-mail: raffaele.cappelli@unibo.it;matteo.ferrara@unibo.it; davide.maltoni@unibo.it).

in emerging countries (e.g., UIDAI project [51]).

There are basically two possibilities to increase the speed of fingerprint identification:

- reducing the total number of fingerprint comparisons (through fingerprint classification [12] [13], pre-filtering or multi-stage matching [5] [6] [9]);
- reducing the processing time (e.g., by designing matching algorithms with lowcomputational complexity [37], or using parallel architectures [47] [26] [30] [22] [2] [46]).

1.2. General-Purpose computing on Graphics Processing Units

A *Graphic Processing Unit* (GPU) is a highly-parallel processor for computer graphics. In response to commercial demand for real-time graphics rendering, GPUs have evolved into many-core processors designed to perform data-parallel computation. The main difference between GPUs and Central Processing Units (CPUs) is that GPUs have proportionally more transistors devoted to arithmetic logic units and less to caches and flow control in comparison to CPUs; GPUs also have higher memory bandwidth compared to CPUs. Recently, the use of GPUs for general-purpose parallel computing is increasingly attracting researchers' interests: General-Purpose computing on Graphics Processing Units (GPGPU) [32] is emerging as a compelling way to deal with computationally demanding tasks, where a large amount of data needs to be processed and/or a large number of operations has to be carried out (e.g., [20] [39] [54]). The parallel processing capability of the GPU allows complex computing tasks to be divided into thousands of smaller tasks that can run concurrently. A typical hardware configuration for GPGPU consists of a CPU (called *host*) connected to one or more GPUs (called *devices*). Given a computation X to be done with GPGPU, X is split into several parts, some of which can be executed in parallel. In order to take effective advantage of the GPU, it is necessary to analyze which parts of X can be executed in parallel on the many processing units of the GPU and write a CPU program (called *host program*) that sends input data and GPU instructions (called *kernel program*s) to the GPU. The GPU executes the given computation in parallel and returns the result to the CPU. To this purpose, specific tools are needed to schedule execution of kernels and communicate with the GPU.

1.3. Contribution of this work

The recent advances in GPU hardware (with a notable growth in computational power and memory capacity) and their successful adoption in many different applications, suggest that GPUs may drastically improve the efficiency of fingerprint identification; this is particularly true for modern local minutiae-matching algorithms, which are well suited for parallel implementations. However, we quickly realized that a simple porting of existing fingerprint recognition algorithms to GPU hardware does not offer relevant advantages. This is also confirmed by the first studies published on this topic by other researchers [22] [2] that just reported minor speed improvements. In fact, the design of an effective GPU fingerprint identification approach requires to address the following issues: (i) limiting data transfer; (ii) using compact data representation (possibly bit-based), (iii) optimizing memory allocation and access; (iv) defining a computation flow that fully exploits the hardware capabilities.

This paper introduces a new parallel algorithm specifically-designed for fingerprint identification on GPUs. The proposed algorithm is based on Minutia Cylinder-Code (MCC) [10] [11], recently introduced as a convenient way to represent fingerprint minutiae. The neighborhood of each minutia is encoded into a fixed-length local structure (called *cylinder*), which can be easily compared to the cylinders obtained from other minutiae neighborhoods. To perform a fingerprint identification, the cylinders of the query fingerprint have to be compared against the cylinders of all database fingerprints. The solution proposed in this paper, thanks to a careful design of the algorithm, ad-hoc data structures and look-up tables, special sorting methods, and many other optimizations, achieves remarkable results. Systematic experiments show that the proposed algorithm can scale up to large databases and achieves a substantial speed-up over: (i) a baseline sequential CPU implementation; (ii) an optimized CPU implementation with SIMD instructions, (iii) the GPU approach in [22] and other state-of-the-art parallel algorithms. A detailed performance analysis of the proposed implementation shows that the execution time is close to the lower bound given by the theoretical maximum instruction throughput of the GPU.

The rest of this paper is organized as follows: Section 2 describes the MCC representation and matching approach; Section 3 introduces a baseline algorithm on CPU and an optimized implementation, which served both as a starting point for designing the GPU algorithm and for measuring the speed-up. Section 4 discusses the optimization goals that guided this study and describes the parallel algorithm. Experimental results and comparison with previous developments are reported in Section 5. Finally, Section 6 draws some conclusions.

2. Minutia Cylinder-Code

This section introduces Minutia Cylinder-Code, briefly describing how minutiae features are encoded into local data structures (the cylinders) and which similarity measures can be used to compare them.

2.1. MCC bit-based representation

Let $MT = \{m_i\}$ be a set of minutiae extracted from a fingerprint: each minutia m is a triplet $m_i = (x_i, y_i, \theta_i)$ where (x_i, y_i) is the minutia location and θ_i is the minutia direction (in the range $[0,2\pi[$). The Minutia Cylinder-Code representation (MCC) [10] associates a local descriptor to each minutia m: this descriptor encodes spatial and directional relationships between the minutia and its neighborhood of radius R, and can be conveniently represented as a cylinder, whose base and height are related to the spatial and directional information, respectively (see Fig. 1.b). The cylinder is divided into N_D sections: each section corresponds to a directional difference in the range $[-\pi, \pi[$; sections are discretized into cells (N_S is the number of cells along the section diameter). During the cylinder creation, a numerical value is calculated for each cell, by accumulating contributions from minutiae in a neighborhood of the projection of the cell center onto the cylinder base. The contribution of each minutia m_t to a cell (of the cylinder corresponding to a given minutia m_i), depends both on:

- spatial information (how much m_t is close to the center of the cell), and
- directional information (how much the directional difference between m_t and m_i is similar to the directional difference associated to the section where the cell lies).

In other words, the value of a cell represents the likelihood of finding minutiae that are close to the cell and whose directional difference with respect to m_i is similar to a given value. Fig. 1.c-d shows the cylinder associated to a minutia with six minutiae in its neighborhood.

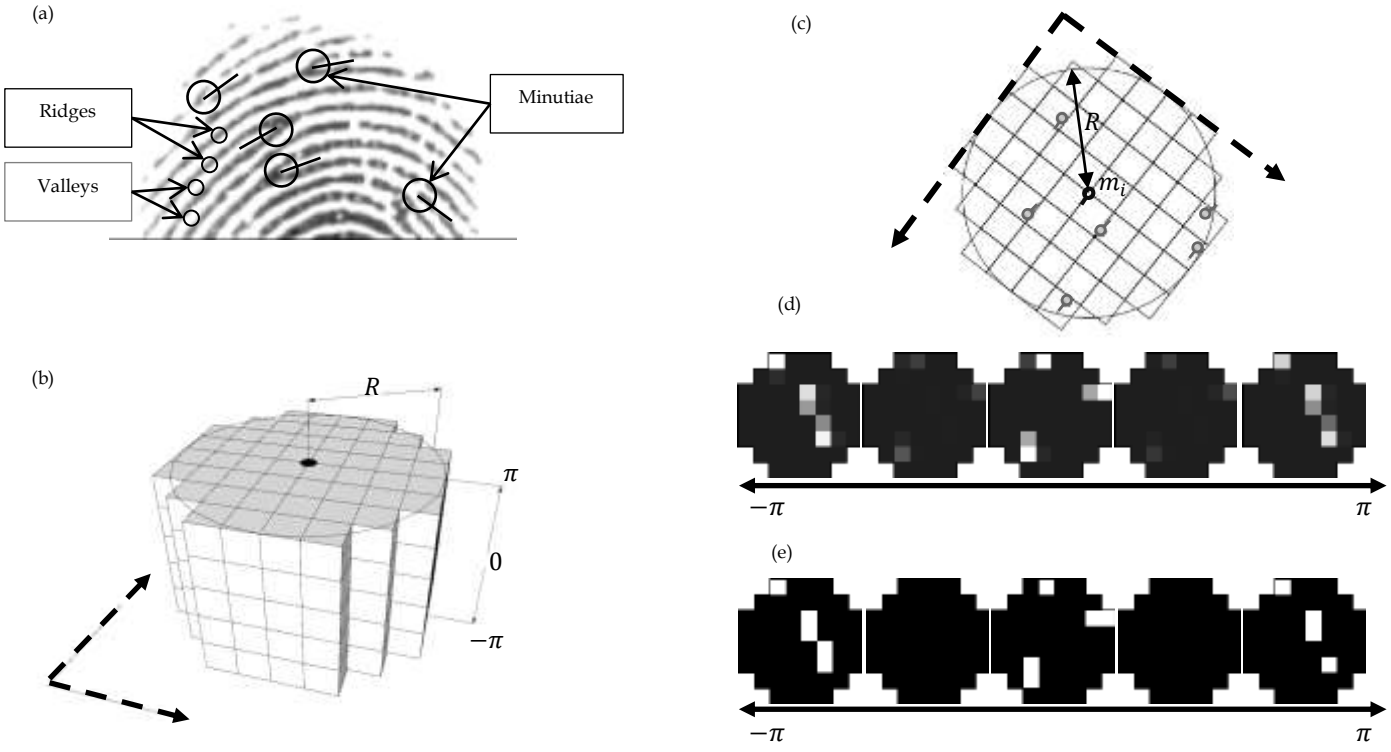

Fig. 1. (a) A fingerprint with ridges, valleys, and minutiae highlighted; (b) a graphical representation of the local descriptor associated to a minutia in the MCC representation, with $N_S = 8$ and $N_D = 5$; (c) minutiae involved in a cylinder; (d) cell values in the N_D sections (lighter areas represent higher values) of the cylinder built over the minutiae in (c); (e) binarized cell values stored as bits. Note that cylinder sections in (d) and (e) are rotated according to the direction of minutia m_i.

Once a cylinder is built from a minutia m_i, it can be simply treated as a single feature vector, obtained by linearizing the cell values. With a negligible loss of accuracy (see [8] and [9]), each element of the feature vector can be stored as a bit (Fig. 1.e): in the following, $\mathbf{v}_i \in \{0,1\}^n$ denotes an MCC bit-vector obtained from minutia m_i, and $T = \{c_i\}$, denotes an

MCC template obtained from a fingerprint, where each $c_i = (\mathbf{v}_i, x_i, y_i, \theta_i)$ is a tuple containing a bit-vector and its associated minutia information. Note that, although strictly speaking the term "cylinder" corresponds to $\mathbf{v}_i$ [10], in the following, for simplicity, it denotes the whole tuple c_i.
Each bit-vector $\mathbf{v}_i$ is a fixed-length local descriptor:

- invariant for translation and rotation, since i) it only encodes distances and directional differences between minutiae, and ii) its base is rotated according to the corresponding minutia angle;
- robust against skin distortion (which is small at a local level) and against small feature extraction errors, thanks to the smoothed nature of the functions defining the contribution of each minutia.

2.2. Similarity measures

As described in [10], a simple but effective similarity measure between two cylinders $c_i = (\mathbf{v}_i, x_i, y_i, \theta_i)$ and $c_j = (\mathbf{v}_j, x_j, y_j, \theta_j)$ is:

$$s_L(c_i, c_j) = \begin{cases} 1 - \frac{\|\mathbf{v}_i \oplus \mathbf{v}_j\|}{\|\mathbf{v}_i\| + \|\mathbf{v}_j\|} & \text{if } d_\phi(\theta_i, \theta_j) \leq \delta_\theta \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where

- $\oplus$ denotes the bitwise XOR operator;
- $\|\cdot\|$ denotes the Euclidean norm;
- $d_\phi(\theta_i, \theta_j)$ is the difference between the two angles;
- δ_θ is a parameter controlling the maximum rotation allowed between two fingerprints.

Note that (1) is a *local* similarity measure between two cylinders: in order to compare two fingerprint templates T_A and T_B, a single value (*global score*), denoting the overall fingerprint similarity, has to be obtained from the pairwise (local) cylinder similarities. Various global similarity measures have been proposed for MCC [10] [11] [8]; the simplest and most efficient one is the *Local Similarity Sort* (LSS), which is calculated as the average of the top n_p local similarity scores between cylinders of the two templates. The value of n_P is not an overall constant, but it partially depends on the number of minutiae in the two templates:

$$n_P = min_{n_P} + \left\lfloor \frac{\left(max_{n_p} - min_{n_p}\right)}{\left(1 + e^{-\tau_P \cdot (\min\{|T_A|, |T_B|\} - \mu_P)}\right)} \right\rfloor \quad (2)$$

where μ_P, τ_P, min_{n_P}, and max_{n_p} are parameters introduced in [10] and $\lfloor \cdot \rceil$ denotes the rounding operator. A more accurate, but less efficient, similarity measure is the *Local Similarity Sort with Distortion-Tolerant Relaxation* (LSS-DTR) [11]. LSS-DTR adds a consolidation step to LSS, in order to obtain a score that reflects to what extent the local similarities hold at global level. LSS-DTR is based on similarity measures between candidate pairs of corresponding minutiae and uses spatial and directional features that are invariant for rotation/translation and tolerate skin distortion.

3. Fingerprint Identification on CPU

Fingerprint identification requires comparing a given fingerprint (query) to all the N fingerprints in a database. Given a database of N MCC templates $DB = \{T_1, T_2, \ldots, T_N\}$ and the template T_Q of a query fingerprint, the result of the $1{:}N$ comparison is a set of N matching scores $S = \{s_1, s_2, \ldots, s_N\}$. Fig. 2 shows a baseline sequential algorithm for fingerprint identification using MCC with LSS, where:

- Each cylinder is simply a pair $c = (\mathbf{v}, \theta)$, since minutiae coordinates x, y are not required by LSS;
- $ComputeNumPairs(\cdot)$ computes the value n_P (2);
- $TopValues(L, n_p)$ returns the top n_p values in L (or L itself if the number of values in L is less than n_p);
- $Sum(L_p)$ computes the sum of the values in L_p.

The algorithm in Fig. 2 can serve as starting point for an optimized implementation. In particular, targeting an x86-64 CPU with SSE extensions [23], the following improvements can be applied.

- Computational complexity: selecting the top n_p local scores is a time consuming operation, with a complexity of $O(nc \cdot log(nc))$, where $nc \approx O(|T_k| \cdot |T_Q|)$ is the number of local similarity scores in L. By quantizing local

similarity scores into w values and adopting a counting-sort strategy [17], the complexity can be reduced to $O(nc + w)$. This approach has the further advantage of reducing the space requirements to a vector $\mathbf{b} \in \mathbb{N}^w$, much smaller than the list L of nc scores. Preliminary experiments showed that, with a proper choice of parameter w (see Table III), this optimization does not compromise recognition accuracy and is much more efficient than other techniques, such as using a specialized partial sorting algorithm [28] with complexity $O\left(nc + n_p \cdot log(n_p)\right)$.

- Optimization of bit vector length: the values chosen for parameters N_s and N_D (see Table III) offer a good trade-off between accuracy and efficiency and result in a bit-vector length $n = 260$ (five sections of 52 bit-valued cells). Preliminary experiments showed that removing one bit from each of the five cylinder sections does not reduce the accuracy and allows $n = 255$, hence each bit-vector $\mathbf{v}$ can be stored in just two 128-bit SSE registers [23].
- Pre-computation of cylinder norms: for each cylinder $c = (\mathbf{v}, \theta)$ of each template, the norm $\eta = \|\mathbf{v}\|$ is calculated once and stored for future use. Hence, each cylinder is a triplet $c = (\mathbf{v}, \theta, \eta)$.
- Bit-wise operations: the bit-wise XOR is computed using SSE instructions [23] operating on 128-bit registers (hence only two SSE instructions are needed to compute $\mathbf{v}_i \oplus \mathbf{v}_j$). The norm of a bit-vector corresponds to the square root of the population count (number of bits with value one), for which an ad-hoc instruction (*popcnt*) is available on recent CPUs [23].
- Look-up table for square roots: since the result of the population-count operation is always an integer number in $[0, n]$, with $n = 255$, it is convenient to use a look-up table ($\mathbf{LUT_{Sqrt}}$) instead of actually computing the square roots.
- Angle quantization: cylinder angles θ are quantized into $z = 256$ values: this allows to implement $d_\theta(\cdot)$ quite efficiently using modulo-256 arithmetic. Note that this does not cause any accuracy degradation since current standards for storing minutiae templates require the same quantization [1].
- A maximum number of cylinders per template is defined ($Max_C = 256$). This allows to precompute any possible output of $ComputeNumPairs(\cdot)$ in look-up table $\mathbf{LUT_p}$. Again, this does not constitute a limitation since current fingerprint minutiae standard templates have the same restriction [1].
- Cylinder norms, $\mathbf{LUT_{Sqrt}}$ values, and match scores are stored as integers using fixed-point arithmetic: this allows to remove any floating-point operation from the algorithm.

Baseline identification algorithm on CPU

Input:
- MCC template T_Q (query)
- N MCC templates $DB = \{T_1, T_2, \dots, T_N\}$

Output:
- N match scores $S = \{s_1, s_2, \dots, s_N\}$

Algorithm:
1. ForEach template T_k in DB
2. $L = new\ List()$; // List of local scores (initially empty)
3. ForEach cylinder $c_i = (\mathbf{v}_i, \theta_i)$ in T_Q
4. ForEach cylinder $c_j = (\mathbf{v}_j, \theta_j)$ in T_k
5. If $d_\phi(\theta_i, \theta_j) \le \delta_\theta$ Then
6. $s_L = 1 - \frac{\|\mathbf{v}_i \oplus \mathbf{v}_j\|}{\|\mathbf{v}_i\| + \|\mathbf{v}_i\|}$ // Computes the local similarity score
7. $L.Add(s_L)$ // Adds the local score to L
8. End If
9. End ForEach
10. End ForEach
11. $n_p = ComputeNumPairs(|T_k|, |T_Q|)$ // scores to consider
12. $L_p = TopValues(L, n_p)$
13. $s_k = Sum(L_p)/n_p$ // Match score between T_Q and T_k
14. End ForEach
15. Return $S = \{s_1, s_2, \dots, s_N\}$

Fig. 2. A simple sequential algorithm for MCC identification on CPU.

Fig. 3 shows the resulting optimized algorithm, where $PopC(\cdot)$ computes the population count of a binary vector using the *popcnt* instruction [23]. The average of the top n_p local similarity scores is computed from $\mathbf{b}$ (lines 11-18), by

accumulating bucket values to reach n_p (the *min* operation in line 14 avoids exceeding n_p with the last bucket value). Note that, to save arithmetic operations, s_k is computed (line 18) as one minus the average of the lowest n_p distances (d_L).

Optimized identification algorithm on CPU
Input:
- MCC template T_Q (query)
- N MCC templates $DB = \{T_1, T_2, \dots, T_N\}$
Output:
- N match scores $S = \{s_1, s_2, \dots, s_N\}$
Algorithm:
1. ForEach template T_k in DB
2. Reset buckets $\mathbf{b}[i], 0 \le i < w$
3. ForEach cylinder $c_i = (\mathbf{v}_i, \theta_i, \eta_i)$ in T_Q
4. ForEach cylinder $c_j = (\mathbf{v}_j, \theta_j, \eta_j)$ in T_k
5. If $d_\phi(\theta_i, \theta_j) \le \delta_\theta$ Then
6. $d_L = \left\lfloor w \cdot \frac{\mathbf{LUT_{Sqrt}}[PopC(\mathbf{v}_i \oplus \mathbf{v}_j)]}{\eta_i + \eta_j} \right\rfloor$ // SSE for bitwise XOR
7. $\mathbf{b}[d_L] = \mathbf{b}[d_L] + 1$ // Increments the corresp. bucket
8. End If
9. End ForEach
10. End ForEach
11. $n_p = \mathbf{LUT_p}[\min\{|T_k|, |T_Q|\}]$ // number of distances to consider
12. $sum = 0$, $t = n_p$, $i = 0$
13. While $i < w$ and $t > 0$ // the sum of the lowest n_p
14. $sum = sum + \min(\mathbf{b}[i], t) \cdot i$ // d_L is computed from $\mathbf{b}$
15. $t = t - \min(\mathbf{b}[i], t)$, $i = i + 1$
16. End While
17. $sum = sum + t \cdot w$ // in case there are less than np votes in $\mathbf{b}$
18. $s_k = 1 - \frac{sum}{np \cdot w}$ // match score between T_Q and T_k
19. End ForEach
20. Return $S = \{s_1, s_2, \dots, s_N\}$

Fig. 3. Optimized sequential algorithm for MCC identification on CPU.

4. Fingerprint Identification on GPU

4.1. CUDA programming model

The *Compute Unified Device Architecture* (CUDA) [43] is one of the most widely-adopted frameworks for GPGPU; CUDA is a hardware and software architecture that enables NVIDIA GPUs to execute parallel kernels written in C/C++. The physical architecture of CUDA-enabled GPUs consists of a set of *Streaming Multiprocessors* (SM), each containing 32 cores for SIMD execution.

In the CUDA programming model, a CUDA kernel is executed in parallel across a set of *threads*, which are organized into *blocks* (Fig. 4). Each thread within a block executes an instance of the kernel and is identified by an index within its block. All threads of the same block are executed on the same SM and share the limited memory resources of that multiprocessor: for this reason the maximum number of threads in a block cannot be too large (1024 in the GPU used in this work, see Table I). However a kernel can be executed by multiple, equally-sized blocks, forming a *grid*: the total number of threads is then equal to the number of blocks times the number of threads per block (Fig. 4) [43]. Each SM schedules and executes threads in groups of 32 parallel threads (being 32 the number of cores in a SM) called *warps*. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp synchronize their execution path. If threads of the same warp take different paths (due to flow control instructions), they have to wait for each other: this situation is called *divergence*. It is important to note that GPU threads are extremely lightweight: typically, thousands of threads are queued up to work in warps of 32 threads each. If the GPU must wait on one warp of threads (for instance due to memory latency or bank conflicts), it simply begins executing work on another warp. Since separate registers are allocated to all active threads, no swapping of registers is needed when switching between two warps: resources remain allocated to each thread until it completes its execution.
CUDA threads have access to various memory types (Fig. 4):

- each thread has its *registers*, which are the fastest memory, and its private *local memory* (which is much slower);
- each block has a small *shared memory*, accessible to all threads of the block and with the same lifetime of the block; shared memory is optimized for 32-bit access and is divided into 32 memory modules, called *banks*, which can be accessed simultaneously; shared memory is very fast if no *bank conflicts* occur (a bank conflict happens when two threads of a warp access two different 32-bit words in the same bank) [42];
- all threads have access to the *global memory*: the largest memory, which is used for communication between different blocks and with the host. The GPU is able to access global memory via 32, 64, or 128-byte memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions, depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads [43]. Therefore a very important optimization in CUDA is ensuring that global memory accesses are as much *coalesced* as possible. In the GPU used in this work (see Table I), an L1 cache for each SM and an L2 cache shared by all SM are present to speed up global memory accesses;
- there is also the possibility of defining a *texture memory space* (a region in global memory which is cached for locality in the *texture cache*, i.e., threads of the same warp reading addresses that are close together achieve better performance); the texture cache is read-only and, within a kernel execution, it is not kept coherent with the underlying global memory. In other words, a kernel can safely read a memory location via texture cache if the location has been modified by a previous kernel execution, but not if it has been modified by the same thread or another one within the same kernel execution;
- finally, all threads have read-only access to the *constant memory space*, a region in global memory that is accessed via *constant cache*, a cache optimized for broadcast, that is access to the same address by all the threads in a warp.

As an example of a specific GPU, Table I reports the main characteristics of the device used in this study.

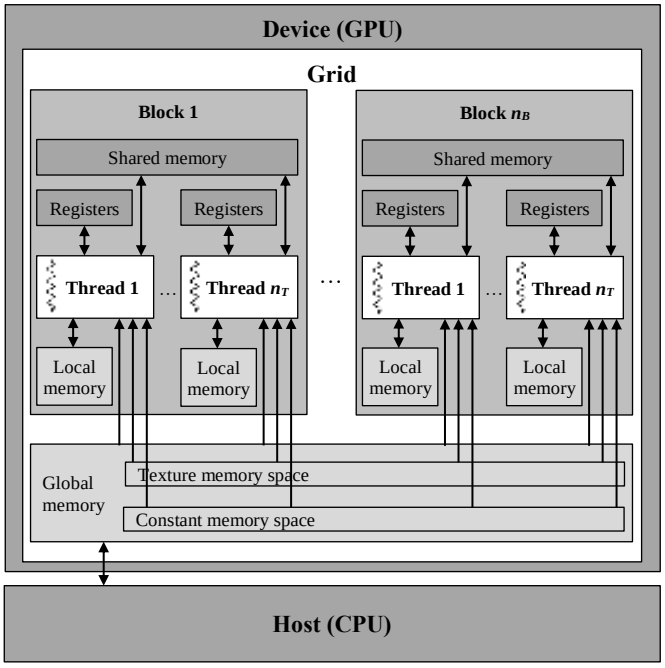

Fig. 4. CUDA: grid, blocks, threads, and the various memory spaces.

4.2. Optimization goals

The design of the parallel algorithm proposed in this work was guided by the following optimization goals:

- OG1 - *Minimize data transfer between the host and the device*. Since the bandwidth between global memory and GPU is much higher than that between host memory and global memory [42], it is very important to minimize the amount of data transferred between the host and the device.
- OG2 - *Choose the most appropriate types of memory and access patterns*. A careful design of all data structures is fundamental: this includes ensuring coalesced access to global memory whenever possible, minimizing bank conflicts

in shared memory, selecting access patterns that maximize cache hits, studying when the use of shared memory, texture cache or constant cache is advantageous.

- OG3 - *Maintain a sufficient number of active threads per multiprocessor*. Executing other warps when one warp is stalled is the only way to hide latencies and keep the GPU hardware busy [42]; the ratio between the number of active warps per SM and the maximum number of possible active warps is called *occupancy* [42]. Low occupancy can interfere with the ability to hide memory latency, resulting in performance degradation.
- OG4 - *Minimize differences in the execution paths within the same warp*. As explained in Section 4.1, divergence causes a performance drop and the kernel code should minimize the number of divergent warps.

Designing an algorithm that achieves the above goals is challenging, not only for the particular paradigm required by GPU programming, but mostly because some of the optimization goals are in contrast with each other and it is necessary to find a good trade-off between them. For example, to hide latency and fulfill OG3, the total number of threads should be increased as much as possible, but for a given computational task X, increasing the number of threads implies that each thread has less operations to do and less data to access in memory: this makes more difficult to fulfill OG2. As another example, see the discussion about database cylinder ordering in Section 4.6: a design that fulfils OG4 does not satisfy OG2, resulting in a performance drop with respect to a design that fulfils OG2 and only partially OG4. A lot of work was necessary to find a good trade-off among these goals: the following section discusses some strategies that were explored.

TABLE I
CHARACTERISTICS OF THE DEVICE USED IN THIS WORK

Device name	Tesla C2075
Architecture	Fermi [44]
Number of SM	14
Total number of cores	448 (14x32)
Clock frequency	1.15 GHz
Max active threads per SM	1536
Max active blocks per SM	8
Max threads per block	1024
Max registers per block	32768
Peak instruction throughput	515 GInstr/s
Global memory	6 GB
Theoretical memory bandwidth	144 GB/s
Max shared memory per SM	16 or 48 KB*
L1 cache size per SM	48 or 16 KB*
L2 cache size	768 KB
Max constant memory space size	64 KB

**The same on-chip memory is used for both L1 cache and shared memory; there are two possible configurations: 48 KB of shared memory and 16 KB of L1 cache or vice versa.*

4.3. Parallelization strategies

Fig. 5 shows a graphical representation of the main computations involved in the optimized sequential algorithm proposed for CPU (Fig. 3). The number of local similarity computations (gray circles in the figure) depends on the number of cylinders in the query template and database templates, while the number of LSS computations (gray rectangles in the figure) depends on the number of database templates N. Looking at Fig. 5, it can be noted that several different strategies may be designed to parallelize the computation. Some possibilities are discussed in the following paragraphs.

- *Strategy A* – A kernel is used to parallelize the computation of local similarities and the host takes care of LSS computation; this strategy is similar to the approach in [22]: a kernel, where each block has to perform a set of local similarity computations, is executed on the device, while LSS is performed on the host. The authors of [22] argue that, given the small number of values to sort, GPU sorting methods do not offer any significant speed-up in this setup. According to our analysis and experiments, this strategy does not fulfill OG1 and cannot achieve high performance: in fact, all local similarities (or at least **b** vectors) have to be copied from the device to the host, while this can be avoided if the LSS computation is carried out on the device.
- *Strategy B* – A kernel is in charge of both local similarities and LSS computation, with one block assigned to each database template. This strategy can fulfill OG1, since only the minimum amount of data needs to be transferred between the device and the host. It can also achieve OG2: in particular, since each block takes care of the computations related to a given database template, shared memory can be effectively exploited to store the vector **b** for the counting-

sort. However, from our preliminary experiments, we understood that with this strategy it is very difficult to find an optimal number of threads per block and to fulfill OG3 and OG4. In fact, since the number of cylinders varies across different database templates and considering that each block must have the same number of threads (possibly multiple of 32, the warp size), it is very difficult to have a sufficient number of active threads, to assign them a balanced workload and, at the same time, to ensure coalesced memory accesses. Moreover, it is likely that during LSS computation many threads remain idle due to the much lower computational complexity of this step with respect to the previous one: this further hinders OG4.

- *Strategy C* – Two different kernels are sequentially executed: the former (named *Step-1*) is in charge of computing local similarities and the latter (named *Step-2*) calculates match scores using LSS. Since two kernels can only communicate via global memory, vectors **b** needs to be stored in global memory by Step-1 and read by Step-2. This is less efficient than storing **b** in shared memory, but has important benefits, since (i) it better balances the thread workload within each kernel (OG4); (ii) guarantees a sufficient number of active threads (OG3), and (iii) optimizes coalesced memory accesses (OG2). In our preliminary studies and experiments, this strategy appeared the most promising one. After several iterations of analysis, implementation, optimization, and performance evaluation, we find out that an effective allocation of the workload is to assign: (i) all computations related to a given database cylinder to a single thread in kernel Step-1 (one "column" of circles in Fig. 5), and (ii) the computation of each match score to a single thread in kernel Step-2 (one rectangle in Fig. 5). The resulting algorithm is described in the following sections.

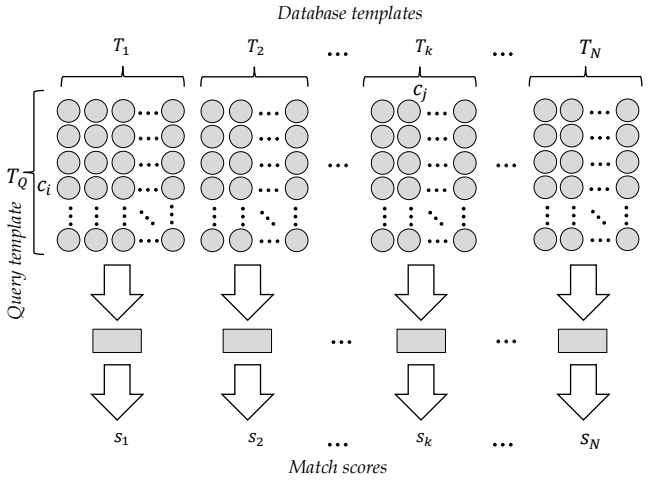

Fig. 5. A graphical representation of the algorithm in Fig. 3: each gray circle represents the computation of a local similarity (between a cylinder c_j of a database template T_k and a cylinder c_i of the query template T_Q) and the increment of the corresponding bucket (lines 5-7 of the algorithm); each gray rectangle represents the computation of a match score s_k between T_Q and T_k using LSS (lines 11-18 of the algorithm).

4.4. The parallel algorithm

Figures 6, 7, 8, and 9 show the parallel algorithm for GPU. With respect to the CPU algorithm in Fig. 3, the following elements have been added:

- A matrix $\mathbf{B} \in \mathbb{N}^{N \times w}$, to store buckets of vectors **b** for all database templates: it allows to compute in parallel local similarities of different database templates.
- A look-up table $\mathbf{LUT_\theta}$, created from the query template T_Q: for any quantized angle θ, $MQ = \mathbf{LUT_\theta}[\theta]$ is the set of cylinder indices in T_Q that are matchable with a cylinder of angle θ: $MQ = \{i | c_i = (\mathbf{v}_i, \theta_i, \eta_i) \in T_Q, d_\phi(\theta_i, \theta) \leq \delta_\theta\}$. This look-up table allows to avoid the conditional statement in line 5 of Fig. 3, easing OG4.
- Each cylinder of a database template is a 4-tuple $c_j = (\mathbf{v}_j, \theta_j, \eta_j, k_j)$, where a new value has been added (k_j, the index of the database template containing c_j).
- Total number of cylinders in the database $NC = \sum_{k=1}^{N} |T_k|$.
- A look-up table $\mathbf{LUT_C}$ containing the number of cylinders for each database template index: $\mathbf{LUT_C}[k] = |T_k|$.
- A matrix $\mathbf{B_s}$, stored in the shared memory of each block of kernel Step-2; it contains a copy of the sub-matrix of **B** that is required by the threads of the block.

The portion of the algorithm executed on the host is reported in Fig. 6: note that it includes only what is executed at each query, it does not include tasks performed only once during system initialization (computing $\mathbf{LUT_{Sqrt}}$, $\mathbf{LUT_p}$, and $\mathbf{LUT}_C$, copying them and the whole DB to the global memory of the device). Line 1 sets all values of $\mathbf{B}$ to zero: no memory is copied from the host to the device, since a CUDA API is used to asynchronously launch a kernel on the GPU that modifies $\mathbf{B}$. The only data transfers between host and device occur at lines 2 and 6; note that only the minimum amount of data is transferred for each query (T_Q to the device and S to the host): this means that OG1 is fulfilled. The rest of the host code consists in the sequential execution of three kernels: ComputeLMC (z threads in one block), Step-1 (NC threads in blocks of BD_1 threads), and Step-2 (N threads in blocks of BD_2 threads). BD_1 and BD_2 should be multiple of 32 (the number of cores in each SM); the choice of their values is discussed in Section 5.1.

MCC Identification on GPU: host program executed on CPU
Input:
- MCC template T_Q (query)
- NC DB cylinders $DB_C = \{c_1, c_2, \dots, c_{NC}\}$ (already in GPU memory)
Output:
- N match scores $S = \{s_1, s_2, \dots, s_N\}$
Algorithm:
1. Reset bucket matrix $\mathbf{B} \in \mathbb{N}^{N \times w}$ on the GPU
2. Copy T_Q to GPU memory
3. Launch Kernel ComputeLMC (z threads per block, 1 block)
4. Launch kernel Step-1 (BD_1 threads per block, $\left\lceil \frac{NC}{BD_1} \right\rceil$ blocks)
5. Launch kernel Step-2 (BD_2 threads per block, $\left\lceil \frac{N}{BD_2} \right\rceil$ blocks)
6. Copy S from GPU memory

Fig. 6. The portion of the algorithm running on the host.

ComputeLMC (Fig. 7) is a simple kernel that initializes $\mathbf{LUT_\theta}$, running one thread for each quantized angle. $\mathbf{LUT_\theta}$ is simply stored as a matrix, with one row for each quantized angle, where the corresponding template indices are sequentially placed in the cells of the row, with a special value denoting the end-of-list (the number of columns is $\left|T_Q\right| + 1$ for the worst case of all cylinders matchable with a given angle, see Table II). Although the workload of this kernel is very small, computing the look-up table on CPU and copying it to the device is slightly less efficient, even with asynchronous memory copy.

MCC Identification on GPU: kernel ComputeLMC executed on GPU
Input:
- MCC template T_Q (query)
- Thread index of the current thread t_{IDX}
Output:
- Matchable query cylinder indices for each discretized angle: $\mathbf{LUT_\theta}$
Kernel execution configuration:
- z threads per block, 1 block (one thread per quantized angle)
Algorithm:
1. $\theta = \frac{2\pi \cdot t_{IDX}}{z}$ // θ is the quantized angle associated to the thread
2. $\mathbf{LUT_\theta}[\theta] = \left\{ i \middle| c_i = (\mathbf{v}_i, \theta_i, \eta_i) \in T_Q, d_\phi(\theta_i, \theta) \le \delta_\theta \right\}$

Fig. 7. The kernel in charge of computing $\mathbf{LUT}_\theta$.

Kernel Step-1 (Fig. 8) executes one thread for each cylinder c_j of the database, in blocks of BD_1 threads: each thread has to compute the local similarity between c_j and all query cylinders whose indices are in $MQ = \mathbf{LUT_\theta}[\theta_j]$, incrementing the corresponding buckets in $\mathbf{B}$. Note that the angles of query cylinders are not needed by this kernel (thanks to $\mathbf{LUT_\theta}$), hence $c_i = (\mathbf{v}_i, \eta_i)$ in line 5. The computation of $PopC(\mathbf{v}_i \oplus \mathbf{v}_j)$, at line 6, is quite efficient on the device, since XOR and population count instructions are natively supported by the GPU [43]. The number of iterations of the ForEach loop at line 4 depends on the number of query cylinders that are matchable with c_j: hence, within a warp, some threads may terminate before others. Although this is not as severe as a divergence, it does not allow a full utilization of the device. To overcome the above problem, one could sort DB cylinders by angle, but in this case the overall efficiency would get worse (see

Section 4.6).

MCC Identification on GPU: kernel Step-1 executed on GPU
Input:
- MCC template T_Q (query)
- Matchable query cylinder indices for each discretized angle: $\mathbf{LUT_\theta}$
- NC DB cylinders $DB_C = \{c_1, c_2, \dots, c_{NC}\}$
- Block index and thread index of the current thread b_{IDX}, t_{IDX}
Output:
- Bucket matrix $\mathbf{B} \in \mathbb{N}^{N \times w}$
Kernel execution configuration:
- BD_1 threads per block, $\left\lceil \frac{NC}{BD_1} \right\rceil$ blocks (one thread per DB cylinder)
Algorithm:
1. $j = BD_1 \cdot b_{IDX} + t_{IDX}$ // j is the DB cylinder index of the thread
2. $c_j = (\mathbf{v}_j, \theta_j, \eta_j, k_j)$ // c_j is the DB cylinder of this thread
   // k_j is the DB template index containing c_j
3. $MQ = \mathbf{LUT_\theta}[\theta_j]$ // the cylinders indices in T_Q matchable with c_j
4. ForEach cylinder index i in MQ
5. $c_i = (\mathbf{v}_i, \eta_i)$ // the i^{th} cylinder in T_Q
6. $d_L = \left\lfloor w \cdot \frac{\mathbf{LUT_{Sqrt}}[PopC(\mathbf{v}_i \oplus \mathbf{v}_j)]}{\eta_i + \eta_j} \right\rfloor$ // Discretized local distance
7. $\mathbf{B}[k_j, d_L] = \mathbf{B}[k_j, d_L] + 1$ // Atomic incr. of the corresp. bucket in $\mathbf{B}$
8. End ForEach

Fig. 8. The kernel in charge of computing local similarities and incrementing the corresponding buckets in **B**.

MCC Identification on GPU: kernel Step-2 executed on GPU
Input:
- Bucket matrix $\mathbf{B} \in \mathbb{N}^{N \times w}$
- Number of cylinders in the query template $|T_Q|$
- Number of cylinders in each DB template $\mathbf{LUT_C}$
- Block index and thread index of the current thread b_{IDX}, t_{IDX}
Output:
- N match scores $S = \{s_1, s_2, \dots, s_N\}$
Kernel execution configuration:
- BD_2 threads per block, $\left\lceil \frac{N}{BD_2} \right\rceil$ blocks (one thread per DB template)
Algorithm:
1. $k_0 = BD_2 \cdot b_{IDX}$ // Index of the first DB template of the block
2. $\mathbf{B_s}[j, t_{IDX}] = \mathbf{B}[k_0 + j, t_{IDX}], 0 \le j < BD_2$ // copy $\mathbf{B}$ sub-matrix to $\mathbf{B_s}$
3. SynchronizeThreads() // barrier for all threads in the block
4. $k = k_0 + t_{IDX}$ // index of the DB template of the current thread
5. $n_p = \mathbf{LUT_p}[\min\{\mathbf{LUT_C}[k], |T_Q|\}]$ // number of distances to consider
6. $sum = 0, \ t = n_p, i = 0$
7. While $i < w$ and $t > 0$ // the sum of the lowest n_p
8. $sum = sum + \min(\mathbf{B_s}[t_{IDX}, i], t) \cdot i$ // d_L is computed from $\mathbf{B_s}$
9. $t = t - \min(\mathbf{B_s}[t_{IDX}, i], t), \ i = i + 1$
10. End While
11. $sum = sum + t \cdot w$ // in case there are less than np votes
12. $s_k = 1 - \frac{sum}{np \cdot w}$ // match score between T_Q and T_k

Fig. 9. The kernel in charge of computing LSS match scores from **B** and storing them in S.

Kernel Step-2 (Fig. 9) executes one thread for each template T_k in the database, in blocks of BD_2 threads: each thread has to compute the LSS match score s_k between T_k and the query template T_Q, starting from values in row k of **B**. All threads in the block cooperate to copy BD_2 rows of **B** (from row k_0) to $\mathbf{B_s} \in \mathbb{N}^{BD_2 \times w}$; note that the pseudo-code in line 2 assumes $w = BD_2$ (as it is in our experiments, see Table III); a barrier synchronization (line 3) is necessary to ensure that the copy has been completed before any thread can continue execution. Lines 5-12 are the same of the optimized CPU algorithm (Fig 3). The number of iterations of the While loop at line 7 depends on n_p and on the distribution of the values in the buckets: hence, each thread has to wait for the thread of the same warp executing the largest number of iterations.

However, according to our experiments, the variance in the number of iterations is quite small and the drop in SM utilization is negligible.

4.5. The data structures

Table II summarizes the data structures of the proposed algorithm, each with the corresponding memory size allocated during the experiments: the main reasons why the particular memory types and data layouts chosen allow to accomplish OG2 are explained in the following.

- DB: to ensure coalesced access at line 2 in Fig. 8, where c_j is read from global memory, cylinders are stored as a *structure of arrays* instead of an *array of structures* (Fig. 10).
- $\mathbf{LUT_{Sqrt}}$, $\mathbf{LUT_{p}}$, $\mathbf{LUT_{\theta}}$, and T_Q are accessed via texture cache; these data structures are read by all the threads with an access pattern that cannot be coalesced but is characterized by high spatial and temporal locality: according to our experiments, using texture cache provides the best performance. This is not obvious for T_Q: in fact, as the query template is the same for all threads, constant cache may seem the best candidate for accessing it. Actually, for implementations not using $\mathbf{LUT_{\theta}}$ optimization, we found that constant cache is the best choice, but when using $\mathbf{LUT_{\theta}}$, the access pattern to cylinders in T_Q becomes more random and texture cache provides better performance than constant cache or shared memory.
- $\mathbf{B}$ and $\mathbf{B_s}$: incrementing buckets in $\mathbf{B}$ (line 7 in Fig. 8) and using them in kernel Step-2 is one of the most critical issues of the algorithm; it is necessary to avoid race conditions when modifying the buckets in global memory and find the most efficient approach to update $\mathbf{B}$ values. We evaluated different solutions, such as using per-thread or per-warp partial copies of $\mathbf{B}$, stored in local or shared memory, to be fused (i.e., summed and stored into global memory) only in a subsequent step (with analogies to the algorithms proposed in [41] [21] [48]). However, according to our findings, thanks to the L2 cache and atomic operations available in the device used [44], if $\mathbf{B}$ is stored in row-major order, and a proper ordering of the database cylinders is chosen (see Section 4.6), the most efficient approach is directly incrementing $\mathbf{B}$ values in global memory through the atomic operation *atomicAdd* [43]. While row-major order is more efficient for kernel Step-1, it would not allow coalesced accesses to $\mathbf{B}$ in kernel Step-2. For this reason, at the beginning of the kernel, the rows of $\mathbf{B}$ that need to be read by threads in the current block are copied (using a coalesced access pattern) to matrix $\mathbf{B_s}$ in shared memory (line 2 in Fig. 9). In this way, there are neither bank conflicts when $\mathbf{B_s}$ is written (line 2), nor when it is read (lines 8-9). In fact, in the former case, at each instruction, the 32 threads of each warp write to 32 consecutive 16-bit words, corresponding to 16 32-bit words of 16 different banks; in the latter, bank conflicts are avoided by padding each $\mathbf{B_s}$ row with $pad_{\mathbf{B}}$ elements, to reach a total size of 33 32-bit words: in our experiments, being each $\mathbf{B_s}$ element a 16-bit word and $w = 64$, we set $pad_{\mathbf{B}} = 2$ (see Table III).
- $\mathbf{LUT_C}$ and S are accessed with a simple coalesced pattern by kernel Step-2, where the 32 threads of each warp read 32 consecutive 16-bit integers from $\mathbf{LUT_C}$ and write 32 consecutive 16-bit integers (scores in fixed-point arithmetic) to S.

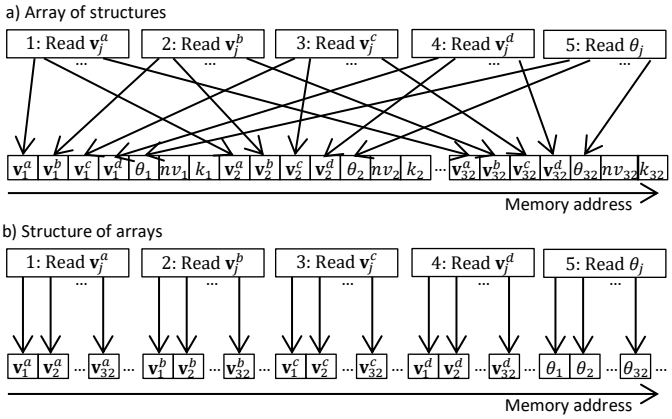

Fig. 10. The effect of DB cylinders data layout on memory coalescing. This example considers five memory read instructions executed by a warp of 32 threads to load bit-vectors $\mathbf{v}_j$ (each stored as four 64-bit words) and their angles θ_j (each stored as a 32-bit word). Layout a) results in 32 memory transactions for each instruction, while layout b) in two transactions for each of the first four instructions and one for the fifth instruction (each transaction reads 128 bytes [43]).

TABLE II
DATA STRUCTURES

Name	Type* [length]	Layout	Actual size with values in Table III	Memory	H↔D transfer	D access
DB	$(BitVect, Int, Int, Int)$ $[NC]$	Structure of arrays	$NC \cdot \left(\left\lceil \frac{n}{8} \right\rceil + 12\right) = 338$ MB	Global	Startup (H→D)	R
$\mathbf{LUT_{Sqrt}}$	Int $[n+1]$	Array	$(n+1) \cdot 4 = 1$ KB	Texture	Startup (H→D)	R
$\mathbf{LUT_p}$	Int $[Max_C]$	Array	$Max_C \cdot 4 = 1$ KB	Texture	Startup (H→D)	R
$\mathbf{LUT_C}$	$ShortInt$ $[N]$	Array	$N \cdot 2 = 488$ KB	Global	Startup (H→D)	R
T_Q	$(BitVect^*, Int, Int)$ $[\lvert T_Q \rvert]$	Structure of arrays	$\lvert T_Q \rvert \cdot \left(\left\lceil \frac{n}{8} \right\rceil + 8\right) = 1.26$ KB $(average)$	Texture	Query (H→D)	R
S	$ShortInt$ $[N]$	Array	$N \cdot 2 = 488$ KB	Global	Query (D→H)	W
$\mathbf{LUT_\theta}$	Int $\left[z \cdot \left(\lvert T_Q \rvert + 1\right)\right]$	Matrix (row-major)	$z \cdot \left(\lvert T_Q \rvert + 1\right) \cdot 4 = 33.3$ KB $(average)$	Texture	Never	R/W†
$\mathbf{B}$	Int $[N \cdot w]$	Matrix (row-major)	$N \cdot w \cdot 4 = 61$ MB	Global	Never	R/W
$\mathbf{B_s}$	$ShortInt$ $[BD_2 \cdot (w + pad_\mathbf{B})]$	Matrix (row-major)	$BD_2 \cdot (w + pad_\mathbf{B}) \cdot 2 = 8.25$ KB	Shared	Never	R/W

* *"BitVect" is a bit-vector (containing n=255 bits) and is stored using four 64-bit words; "Int" is a 32-bit word; "ShortInt" is a 16-bit word.*
† $\boldsymbol{LUT_\theta}$ *is written by a kernel (ComputeLMC) that directly modifies the underlying global memory, then it is read through the texture cache by another kernel (Step-1).*

4.6. Effects of database cylinder ordering

Kernel Step-1 considers each database cylinder separately (one for each thread) and any database cylinder $c_j = \left(\mathbf{v}_j, \theta_j, \eta_j, k_j\right)$ contains the index k_j of the template it belongs to. For this reason, cylinders c_j can be sorted in any order and it is therefore worthwhile studying which order provides the best efficiency. It should be noted that:

- ordering by θ_j minimizes the risk of different number of iterations in the loop of kernel Step-1 (line 4 in Fig. 8), hence improving OG4;
- ordering by k_j maximizes L2 cache hits in the atomic increment of **B** elements by kernel Step-1 (line 7 in Fig. 8), hence improving OG2 (in the device used, the L2 cache helps to accelerate atomic operations, drastically reducing the number of write backs [44]).

According to our experiments, the most efficient trade-off is ordering first by k_j and then, within the same template, by θ_j. Furthermore, while it is desirable that threads of the same warp tend to access the same row of **B**, we found that accesses to the same row by threads belonging to different warps of the same block reduce the efficiency, probably because warps cannot be swapped fast enough due to the dependency on the data values of the collided memory locations. For this reason, after sorting the database cylinders as described above, a mapping function $\mathcal{M}_c: \mathbb{N} \to \mathbb{N}$ is applied, reordering the cylinders so that c_j is moved to position $\mathcal{M}_c(j)$.

$$\mathcal{M}_c(j) = 32 \cdot \left(\frac{NC}{BD_1} \cdot W_j + B_j\right) + j_W \tag{3}$$

where:

- $B_j = \left\lfloor \frac{j}{BD_1} \right\rfloor$ is the index of the block of the j-th thread;
- $j_B = j \bmod BD_1$ is the index of the j-th thread in block B_j;
- $W_j = \left\lfloor \frac{j_B}{32} \right\rfloor$ is the index of the warp of the j-th thread in block B_j;
- $j_W = j \bmod 32$ is the index of the j-th thread in warp W_j.

The mapping function (3) separates cylinders accessed by different warps of the same block, while keeping together cylinders accessed by threads of the same warp. Fig. 11 shows a graphical example of the effect of the mapping function.

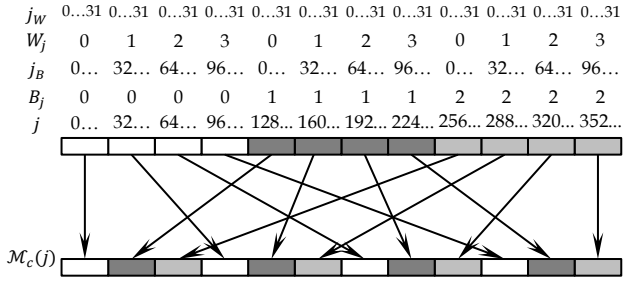

Fig. 11. Effect of the mapping function (3) in a simplified case with $NC = 384$, $BD_1 = 128$; each rectangle represents 32 cylinders.

5. Performance

The proposed parallel algorithm was evaluated on a PC with Intel Xeon CPU running at 2.0 GHz and a NVIDIA Tesla C2075 GPU (see Table I). The baseline CPU algorithm was implemented in C#, the optimized CPU algorithm in C++ with compiler intrinsics [23] to generate SSE and *popcnt* instructions. The proposed algorithm was implemented using C# and C++ for the host code and CUDA C [43] for the device code. The following sections report and discuss experiments carried out to measure the performance of the proposed algorithm, determine the speed-up over the baseline algorithm, and compare the results to previous works published in the literature.

TABLE III
PARAMETER VALUES USED IN THE EXPERIMENTS

Parameter*	Description	Value
N_D	Sections in each cylinder	5
N_S	Cells along the diameter of each section	8
n	Bits in each cylinder (bit-vector length)	255
min_{n_p}	Parameters for computing LSS (2)	11
max_{n_p}		13
μ_P		30
τ_P		2/5
δ_θ	Max allowed rotation between two fingerprints	$\pi/6$
w	Possible values for quantized local similarities	64
z	Possible values for quantized angles	256
Max_C	Maximum number of cylinders per template	256
BD_1	Threads per block in kernel Step-1	192
BD_2	Threads per block in kernel Step-2	64
$pad_{\mathbf{B}}$	Padding elements for each row in $\mathbf{B}_s$	2

* *For any MCC parameter not listed in this table, the same value of [11] was used.*

TABLE IV
NUMBER OF TEMPLATES AND CYLINDERS IN THE BENCHMARK

Description	Value
Number of DB templates (N)	250 000
Total number of DB template cylinders (NC)	8 056 696
Number of query templates	100 000
Average number of cylinders in a query template ($\lvert T_Q \rvert$)	32.3

5.1. Benchmark and parameters

Although several fingerprint databases are available for evaluating fingerprint verification algorithms (1:1 comparison), e.g., [34] [35] [36] [7] [18], they are too small for assessing the performance of large-scale identification systems (1:N comparison). For this reason we decided to create a new benchmark combining different sources: (i) FVC2004 DB1 [36], (ii) BioSec FO database [52], (iii) FingerPass DB2 [27], (iv) Synthetic fingerprints generated by SFinGe [15].

The three above real databases were acquired using optical sensors at 500 dpi, and the synthetic generator was tuned to synthetize images with the same resolution. The resulting benchmark consists of (see Table IV):

- a set of 250 000 fingerprints (to be used as DB) containing 400 real impressions and 249 600 synthetic ones;
- a set of 100 000 fingerprints (to be used for simulating queries); the first 50 000 (400 real) with a mate in DB, the other 50 000 (400 real) without mate in DB.

After minutiae extraction and creation of MCC cylinders, the MCC template of each fingerprint was stored on disk and used for the experiments reported in the rest of this section. The parameter values reported in Table III were used for all experiments. The choice of the number of threads per block for kernel Step-1 and Step-2 (BD_1 and BD_2) was made taking into account the following constraints:

- BD_1 and BD_2 must be multiple of 32 (number of cores per SM), to avoid wasting computational resources;

- the device limits cannot be exceeded (max active blocks per SM, max active threads per SM, max threads per block, max registers per SM, max shared memory per SM), see Table I;
- the number of registers per thread required by the kernels (24 in kernel Step-1, 30 in kernel Step-2);
- the amount of shared memory required by kernel Step-2, which depends on BD_2, see Table II.

The values of BD_1 and BD_2 in Table III maximize the number of active warps per SM (OG3) while satisfying the above constraints.

5.2. Execution time and scalability

Table V reports the total execution time of the first ten[1] queries in the benchmark, using the two sequential CPU algorithms (baseline and optimized) and the proposed parallel algorithm. For each algorithm, the average throughput is measured as the number of fingerprint comparisons per unit of time (matches per second). From the table it is well evident that the optimizations described in Section 3 (from the baseline to the optimized CPU algorithm) are very effective: the execution time is reduced by more than nine times. Even more impressive is the result of the parallel algorithm: the execution time is further shortened by more than 50 times.

Table VI provides more insights on the proposed parallel algorithm, showing the execution times of the various steps (see Fig. 6): this information was obtained using the NVIDIA Visual Profiler [45]. Note that step a) is executed asynchronously with respect to steps b) and c); however, in our experiments, only step b) is actually executed in parallel to a). This is because step a) fully utilizes the GPU cores and can be overlapped only to memory-transfer operations like b), while c) can only start when a) is completed. Steps d)-f) are executed sequentially, as described in Section 4.4, while g) represents various operations that the CPU has to execute, such as configuring kernel parameters and launching kernels. As expected, the execution time is dominated by kernel Step-1, which is the most computationally-intensive task. However, it is worth noting that the very small execution time of kernel Step-2 is the result of very careful parallelization strategies and optimizations. In the first parallelization attempts (for instance using strategies A or B in Section 4.3), the time needed for the same task was ten to twenty times higher. Since most of the execution time is due to kernel Step-1, a specific analysis was performed on it and reported in Appendix A.

TABLE V
TIME NEEDED FOR THE FIRST TEN QUERIES AND CORRESPONDING THROUGHPUT (THOUSAND MATCHES PER SECOND)

Algorithm	Execution time (ms)	Throughput (KMPS)
Baseline algorithm on CPU	143114.83	17.5
Optimized algorithm on CPU	15724.81	159.0
Proposed parallel algorithm on GPU	277.68	9003.2

TABLE VI
TOTAL EXECUTION TIME OF THE FIRST TEN QUERIES OF THE BENCHMARK, BROKEN DOWN INTO THE MAIN STEPS

Step	Time (ms)	Percent
a) Reset bucket matrix **B**	4.63	1.67%
b) Copy T_Q to the GPU memory *	0.04	-
c) Kernel ComputeLMC	0.32	0.12%
d) Kernel Step-1	259.59	93.49%
e) Kernel Step-2	9.46	3.41%
f) Copy S from GPU memory	0.78	0.28%
g) Other CPU activities	2.9	1.04%
Total execution time	277.68	100.00%

**This does not contribute to the total time since step b) is executed in parallel with a).*

While previous experiments used the maximum DB size in the benchmark ($N = 250000$), Fig. 12 reports the results of a scalability experiment, where the same queries were performed on databases with size N varying from 10 to 250 000. If the size of the database is small ($N < 1000$), the throughput of the proposed algorithm is much lower, although still higher than the baseline and optimized CPU algorithms. At larger DB sizes ($1000 \leq N < 10000$), the advantage of the

[1] The experiments reported in this section and in the following one required several repetitions: for this reason they were carried out on a small number of queries; experiments running all queries in the benchmark are described in Appendix A.

proposed algorithm is evident, and for $N > 10000$, its throughput remains steady at 9 million matches per second: no scalability issues were found.

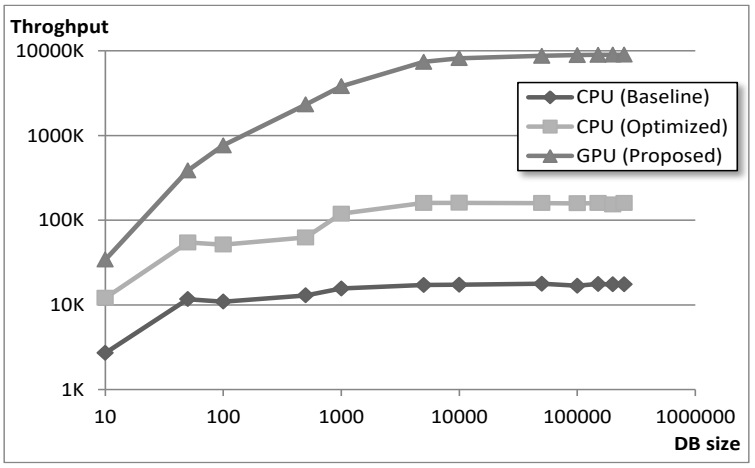

Fig. 12. Average throughput (matches per second) of the first ten queries in the benchmark varying N from 10 to 250 000. Logarithmic scales are used for both axes to better show the trends.

5.3. Impact of some optimizations

Various experiments were carried out to understand the impact of the optimizations introduced in the parallel algorithm. Table VII summarizes the results: the first row reports the throughput of the algorithm as it is proposed; rows 2-6 show the effects of removing, individually, five optimizations from kernel Step-1; rows 7-9 show the effects of removing, individually, three optimizations from kernel Step-2; the last row reports the throughput without all such optimizations.

TABLE VII
IMPACT OF SOME OPTIMIZATIONS: THROUGHPUT (MILLIONS OF MATCHES PER SECOND) MEASURED ON THE FIRST TEN QUERIES

Optimizations	Throughput (MMPS)	Performance drop
All proposed optimizations	9.003	-
No c_j ordering as in Section 4.6	8.250	8.4%
T_Q not accessed via texture cache	7.089	21.3%
$\mathbf{LUT_{Sqrt}}$ not accessed via texture cache	8.827	2.0%
$\mathbf{LUT_{\theta}}$ not accessed via texture cache	7.495	16.8%
DB stored as array of structures	8.685	3.5%
No padding in $\mathbf{B_s}$ ($pad_{\mathbf{B}} = 0$)	8.574	4.8%
No $\mathbf{B_s}$ (Kernel Step-2 uses $\mathbf{B}$)	7.603	15.6%
$\mathbf{LUT_p}$ not accessed via texture cache	8.991	0.1%
None of the optimizations above	5.716	36.5%

From the results in Table VII and the corresponding execution times, the following observations can be drawn:

- the cylinder ordering scheme discussed in Section 4.6 reduces the execution time of kernel Step-1 of more than 8% and allows to gain about 750 KMPS of throughput;
- the use of texture cache has a noticeable impact on performance, especially for T_Q and $\mathbf{LUT_{\theta}}$;
- storing DB as a structure of arrays (to allow coalesced memory read) helps, but less than expected considering that non-coalesced accesses results in many more memory transactions. This is due to the nature of kernel Step-1, which is instruction-bound (see Appendix A), and has enough computation instructions to hide most of the higher memory latencies due to non-coalesced accesses.
- if no padding is used to minimize shared bank conflicts in kernel Step-2, the total time required by the execution of this kernel over the ten queries grows from 9.46 ms (Table VI) to 13.89 ms (about 47% more);
- if $\mathbf{B_s}$ is not used, kernel Step-2 makes non-coalesced accesses to $\mathbf{B}$ and this results in a 15.6% overall performance drop, with a huge growth of the execution time of this kernel (from 9.46 ms to 51.13 ms). In fact, being this kernel memory-bound, coalesced memory-access is fundamental.

5.4. Testing within a complete identification system

In order to experiment the proposed algorithm in a realistic scenario, the prototype of a complete identification system was implemented. The system consists of three main modules:

- *Identification Query Manager* (IQM) – maintains a queue of the queries received from the clients (each with the corresponding T_Q); submits each request to the Identification Module; provides the results to the clients;
- *Identification Module* (IM) – performs 1:N identification of T_Q against the templates in DB using MCC with LSS; provides match scores S to the Refinement Module;
- *Refinement Module* (RM) – considers the DB templates corresponding to the top 0.1% scores in S and compares them to T_Q using MCC with LSS-DTR (see Section 2.2); provides the final result to IQM.

IQM and RM run on the host and are implemented in C# and C++, using the Task Parallel Library (TPL) [29] to easily parallelize IQM and RM activities on the CPU. Four implementations of IM have been tested: two running on the host (using the baseline and the optimized CPU algorithm, respectively), and two using the proposed GPU algorithm (one on a single GPU and one dynamically dividing the workload among four GPUs installed on the same PC).

The full benchmark described in Section 5.1 was used in the test: $N = 250\,000$ templates in DB and 100 000 queries (50 000 with a mate in DB and 50 000 without mate). Testing with both mated and non-mated queries is important to evaluate the biometric accuracy of the identification system, which can be assessed with two performance indicators [51]:

- False Negative Identification Rate (FNIR) – percentage of queries with a mate in DB that are not correctly identified;
- False Positive Identification Rate (FPIR) – percentage of queries without a mate in DB that are mistakenly identified.

The same accuracy was obtained by all the algorithms experimented in IM: FNIR=1.25% and FPIR=0%. This level of accuracy is in line with large-scale identification results using a single finger [51] and with the accuracy of state-of-the-art fingerprint verification algorithms [18].

Table VIII reports the total time required by the system to complete all the queries and the corresponding throughput (thousands of matches per second). In the single GPU case, the throughputs are slightly higher than those measured on the first ten queries (Table V) because the average number of query cylinders on the whole benchmark (32.3) is smaller than that on the first ten queries (41.5). With the proposed GPU algorithm in IM, the system completes all the queries in about 12 minutes using four GPUs and in 45 minutes with one GPU; more than one day is necessary with the optimized CPU algorithm, and more than 15 days with the baseline one.

TABLE VIII
COMPLETE IDENTIFICATION SYSTEM PROTOTYPE: TOTAL EXECUTION TIME AND THROUGHPUT FOR THE WHOLE BENCHMARK

Algorithm used in IM	Execution time (dd:hh:mm)	Throughput (KMPS)
Baseline algorithm on CPU	15:23:40	18.1
Optimized algorithm on CPU	01:16:50	170.1
Proposed algorithm on one GPU	00:00:45	9 304.6
Proposed algorithm on four GPUs	00:00:12	35 221.4

5.5. Comparison with previous works

The efficiency of the proposed algorithm was compared to results published in the literature for other parallel fingerprint-identification systems based on the main hardware architectures (FPGA, CPU, GPU). In particular, the following studies were considered:

- Lindoso et al. (2007) [30], a FPGA-based fingerprint identification system, whose speed-up is measured with respect to a sequential algorithm running on a 3 GHz Intel Pentium IV processor;
- Jiang and Crookes (2008) [26], a FPGA-based fingerprint identification system; to the best of our knowledge this is the fastest FPGA-based algorithm reported in the scientific literature; its speed-up is measured with respect to a sequential algorithm running on a 2.8 GHz Intel Celeron processor;
- Peralta et al. (2014) [46], a CPU-based fingerprint identification system running on a cluster of 12 nodes, each equipped with two Intel Xeon E5-2620 processors (each node can run up to 24 parallel threads); three different algorithms were experimented in this study, but for simplicity in the following comparison only the fastest one is reported; the speed-up is measured with respect to a sequential execution of the same algorithm on a single CPU core;
- Gutierrez et al. (2014) [22], a GPU-based fingerprint identification system; to the best of our knowledge, this is the only study published on scientific journals that reports fingerprint identification results on GPUs; results from all the

three different GPU hardware configurations experimented in this study are reported in the following comparison; the speed-up is measured with respect to a sequential algorithm running on an Intel Xeon E5-2630 processor.

Table IX compares the performance of the above systems to the proposed algorithm; each row of the table reports:

- The size of the fingerprint database on which identification experiments were carried out;
- The throughput (in thousands of fingerprints matched per second) of the sequential algorithm on CPU used to calculated the speed-up of the respective parallel algorithm, as reported in the corresponding paper;
- A short description of the specific hardware used for the parallel algorithm;
- The maximum number of threads that the hardware can execute in parallel (where applicable);
- The throughput (in thousands of fingerprints matched per second) of the parallel algorithm, as reported in the corresponding paper;
- The speed-up, simply measured as the ratio between the two throughputs.

It is well evident that the proposed GPU algorithm overcomes all previous approaches, both in terms of absolute performance and in terms of relative speed-up. In particular, the following observations can be drawn.

- Using just one GPU, the proposed algorithm is able to compare more than nine millions of fingerprints per second: it is two orders of magnitude faster than the GPU algorithm described in [22] and one order of magnitude faster than a cluster of 12 PCs running the algorithm described in [46]. On a PC with four GPUs the throughput is more than 35 millions of fingerprints per second: with such a throughput, less than 0.3 seconds would be required to perform ten queries on a database with one million fingerprints.
- The speed-up of the proposed algorithm is remarkable, especially if compared to the results in [22], which were obtained on GPUs with more computational power than those used in this work; for instance, on a GPU with 512 cores, [22] obtained a 50x speed-up, while the proposed algorithm shows a 514x speed-up on a GPU with 448 cores.
- On a single GPU, the speed-up of the proposed algorithm over the optimized CPU implementation is 54.7x: even if the optimized CPU algorithm were considered as the reference, the speed-up would be higher than those of both single-GPU results in [22].

TABLE IX
COMPARISON WITH PREVIOUSLY PUBLISHED PARALLEL ALGORITHMS ON FPGAS, CPUS, AND GPUS.

Method	DB size	Throughput of the baseline algorithm on CPU (KMPS)	Hardware	Parallel threads	Throughput of the parallel algorithm (KMPS)	Speed-up
Lindoso et al. (2007) [30]	56	0.3	FPGA Xilinx Virtex-4 LX	-	7.1	23.7
Jiang and Crookes (2008) [26]	10 000	26.2	FPGA Xilinx Virtex-E	-	1 219.5	46.5
Peralta et al. (2014) [46]	400 000	4.5	Cluster of 12 nodes based on Intel Xeon CPU E5-2620, 12 cores (24 threads) per node	288	812.7	180.6
Gutierrez et al. (2014) [22]	100 000	1.6	One GeForce GTX 680 GPU	1536	55.7	34.8
			One Tesla M2090 GPU	512	50.0	31.3
			Two Tesla M2090 GPUs	1024	97.7	61.1
Proposed algorithm	250 000	18.1	One Tesla C2075 GPU	448	9 304.6	514.1
			Four Tesla C2075 GPUs	1792	35 221.4	1 945.9

6 Conclusions

In this paper we introduced a new parallel algorithm for fingerprint identification optimized for GPUs. A careful design of the algorithm, data structures and memory usage allows the raw computational power of the GPU used to be fully exploited(the performance measured is close to the theoretical maximum).

In our experiments on a PC with four GPUs, the prototype of a complete identification system achieved a throughput of more than 35 million fingerprint matches per second, more than two orders of magnitude higher than the best results reported for previously published GPU algorithms [22] [2]. With such a throughput, less than 0.3 seconds are required to perform ten queries on a database with one million fingerprints, thus enabling real-time fingerprint identification on large databases with low-cost hardware.

Even if the optimizations described in this work are specific to the MCC fingerprint recognition algorithm, we believe that the optimization goals defined and the overall parallelization strategy adopted can be successfully applied for porting

to GPU a wider class of pattern recognition algorithms, in particular those based on local feature matching, such as SIFT [31] and SURF [3].

In the future we plan to study multi-core CPU and multi-GPU solutions for multimodal biometric recognition (e.g., combining fingerprint and iris biometric traits).

7 Acknowledgements

This work was supported by the European Community's Framework Programme (FP7/2007-2013) under Grant agreement 284862.

Appendix A: Analysis of the execution time of kernel Step-1

Let Ω be the total number of local similarities computed during an execution of kernel Step-1. The total amount of global memory bytes accessed by the kernel is:

$$M_{RW} = NC \cdot 44 + (\Omega + NC) \cdot 4 + \Omega \cdot 36 + \Omega \cdot 4 + \Omega \cdot 8 \quad (4)$$

The five terms of the sum in (4) correspond to (see also Table II): (i) reading all DB cylinders, (ii) reading values from $\mathbf{LUT_{\theta}}$ (including the special value denoting the end-of-list, which each of the NC threads has to read before exiting its ForEach loop), (iii) reading a query cylinder each time a local similarity has to be computed, (iv) reading $\mathbf{LUT_{Sqrt}}$ values, and (v) incrementing values in $\mathbf{B}$.

An analysis of the assembly code of the kernel shows that the number of arithmetic and control flow instructions executed by a thread is $NI_T \cong 31 + \omega \cdot 61$, where ω is the number of iterations in the ForEach loop (line 4 in Fig. 8). The total number of arithmetic and control flow instructions executed by the kernel is $NI_K \cong NC \cdot (31 + \bar{\omega} \cdot 61)$, where $\bar{\omega}$ is the average number of iterations, among all warps, in the ForEach loop.

In the following, the above equations are used to analyze the execution time of the first query of the benchmark. In such query, $\Omega = 70\,965\,592$; then $M_{RW} \cong 3.8$ GB, considering that $NC = 8\,056\,696$ (Table IV). Note that M_{RW} is the total number of global memory bytes requested to be read or written by the kernel: the actual amount of global memory accessed can be very different, due to non-coalesced memory accesses (which increase it), and caching (which decreases it). According to the profiler [45], the actual amount of global memory accessed during the query is $M'_{RW} = 1.22$ GB, about one third of M_{RW}, confirming that the proposed algorithm and data structures fulfill OG2. For the same query, $\bar{\omega} = 16.14$, then $NI_K \cong 8.2 \cdot 10^9$. The ratio $R_K = NI_K : M'_{RW}$ is about 6.7; this ratio is useful to understand if the kernel is memory-bound or instruction-bound, that is if its performance is limited by memory throughput or by instruction throughput, respectively [38]. To this purpose, R_K is compared to the ratio R_P between peak instruction throughput and peak memory throughput of the device: kernels with $R_K < R_P$ are usually memory-bound, kernels with $R_K > R_P$ are usually instruction-bound [38]. For the GPU used in this study, $R_P = 515 : 144 \cong 3.6$ (see Table I), hence kernel Step-1 is definitely instruction-bound[2]. The execution time of an instruction-bound kernel is limited by the maximum instruction throughput of the device, which varies according to the instruction type (e.g., an integer add requires one clock cycle, while an integer multiply requires two cycles [43]). An analysis of the assembly code showed that the mix of instructions in kernel Step-1 requires, on the average, $CPI = \frac{1.58 + \bar{\omega} \cdot 1.48}{\bar{\omega} + 1}$ clock cycles per instruction, which is about 1.5 for any reasonable value of $\bar{\omega}$, including that of the query we are considering. The theoretical lower bound to the execution time of the kernel, assuming that all memory latencies are hidden by computation, and disregarding any warp- or block-scheduling latencies, is $T_{LB} \cong \frac{NI_K \cdot CPI}{32 \cdot N_{SM} \cdot F_C}$, where N_{SM} is the number of SM and F_C the GPU clock frequency. Since $N_{SM} = 14$ and $F_C = 1.15$ GHz (see Table I), $T_{LB} \cong 23.9$ ms. T_{LB} represents the minimum possible execution time on the given GPU hardware. The actual execution time of the kernel, for the query considered, is 27.1 ms. In conclusion, our execution time is just 13% higher than the theoretical hardware limit T_{LB}, thus confirming that we are effectively exploiting the computational power of the device.

[2] An example of memory-bound kernel is kernel Step-2, which has a ratio $R_K \cong 2.2$.

References

[1] ISO/IEC 19794-2, Information technology -- Biometric data interchange formats -- Part 2: Finger minutiae data, 2011.

[2] A.I. Awad, "Fingerprint local invariant feature extraction on GPU with CUDA," *Informatica (Slovenia)*, vol. 37, no. 3, pp. 279-284, 2013.

[3] H. Bay, A. Ess, T.M Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346-359, 2008.

[4] J. Bringer and V. Despiegel, "Binary Feature Vector Fingerprint Representation from Minutiae Vicinities," in *IEEE 4th International Conference on Biometrics Theory, Applications and Systems*, Washington, 2010.

[5] R. Cappelli, "Fast and accurate fingerprint indexing based on ridge orientation and frequency," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 41, no. 6, pp. 1511-1521, December 2011.

[6] R. Cappelli and M. Ferrara, "A fingerprint retrieval system based on level-1 and level-2 features," *Expert Systems With Applications*, vol. 39, no. 12, pp. 10465-10478, September 2012.

[7] R. Cappelli, M. Ferrara, A. Franco, and D. Maltoni, "Fingerprint verification competition 2006," *Biometric Technology Today*, vol. 15, no. 7-8, pp. 7-9, August 2007.

[8] R. Cappelli, M. Ferrara, and D. Maio, "A Fast and Accurate Palmprint Recognition System based on Minutiae," *IEEE Transactions on Systems, Man and Cybernetics - Part B*, vol. 42, no. 3, pp. 956-962, June 2012.

[9] R. Cappelli, M. Ferrara, and D. Maltoni, "Fingerprint Indexing based on Minutia Cylinder Code," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 5, pp. 1051 - 1057, May 2011.

[10] R. Cappelli, M. Ferrara, and D. Maltoni, "Minutia Cylinder-Code: a new representation and matching technique for fingerprint recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 12, pp. 2128 - 2141, December 2010.

[BO] R. Cappelli, M. Ferrara, and D. Maltoni, "Minutiae-Based Fingerprint Matching," in *Cross Disciplinary Biometric Systems*, Mago Vijay Kumar Liu Chengjun, Ed.: Springer, 2012.

[11] R. Cappelli, M. Ferrara, D. Maltoni, and M. Tistarelli, "MCC: a Baseline Algorithm for Fingerprint Verification in FVC-onGoing," in *proceedings 11th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, Singapore, 2010.

[12] R. Cappelli and D. Maio, "The State of the Art in Fingerprint Classification," in *Automatic Fingerprint Recognition Systems*.: Springer, 2004, ch. 9, pp. 183-205.

[13] R. Cappelli, D. Maio, and D. Maltoni, "Combining fingerprint classifiers," in *First International Workshop on Multiple Classifier Systems (MCS2000)*, Cagliari, 2000, pp. 351-361.

[15] R. Cappelli, D. Maio, and D. Maltoni, "Synthetic Fingerprint-Database Generation," in *16th International Conference on Pattern Recognition (ICPR2002)*, Québec City, 2002, pp. 744-747.

[16] Xinjian Chen, Jie Tian, and Xin Yang, "A new algorithm for distorted fingerprints matching based on normalized fuzzy similarity measure," *IEEE Transactions on Image Processing*, vol. 15, no. 3, pp. 767-776, March 2006.

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Counting Sort," in *Introduction to Algorithms*.: MIT Press and McGraw-Hill, 2001, ch. 8.2, pp. 168–170.

[18] B. Dorizzi et al., "Fingerprint and On-line signature Verification Competitions at ICB 2009," in *Proceedings 3rd IAPR/IEEE International Conference on Biometrics (ICB09)*, Alghero, 2009.

[19] M. Ferrara, D. Maltoni, and R. Cappelli, "Noninvertible Minutia Cylinder-Code Representation," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 6, pp. 1727-1737, December 2012.

[20] M. Fort and J.A. Sellarès, "Solving the k-influence region problem with the GPU," *Information Sciences*, vol. 269, pp. 255-269, June 2014.

[21] J. Gomez-Luna, J.M. Gonzalez-Linares, J.I. Benavides Benitez, and N. Guil Mata, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 11, pp. 2273-2282, Nov. 2013.

[22] P.D. Gutierrez, M. Lastra, F. Herrera, and J.M. Benitez., "A high performance fingerprint matching system for large databases based on GPU," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 62-71, January 2014.

[23] Intel. (2014, October) Intel Instruction Set Architecture Extensions. [Online]. http://software.intel.com/en-us/intel-isa-extensions

[24] A K Jain, P Flynn, A A Ross, and SpringerLink, *Handbook of Biometrics*.: Springer, 2008.

[25] Lakhmi C. Jain et al., Eds., *Intelligent Biometric Techniques in Fingerprint and Face Recognition*.: CRC Press, 1999.

[26] R.M. Jiang and D. Crookes, "FPGA-based minutia matching for biometric fingerprint image database retrieval," *Journal of Real-Time Image Processing*, vol. 3, no. 3, pp. 177-182, September 2008.

[27] X. Jia, X. Yang, Y. Zang, N. Zhang, and J. Tian, "A cross-device matching fingerprint database from multi-type sensors," in *proceedings of 21st International Conference on Pattern Recognition (ICPR)*, 2012, pp. 3001-3004.

[28] Donald Knuth, *The Art of Computer Programming*, 3rd ed.: Addison-Wesley, 1997.

[29] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," *SIGPLAN Not.*, vol. 44, no. 10, pp. 227-242, October 2009.

[30] A. Lindoso, L. Entrena, and J. Izquierdo, "FPGA-Based Acceleration of Fingerprint Minutiae Matching," in *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, Mar del Plata, Argentina, 2007, pp. 81-86.

[31] D.G. Lowe, "Object recognition from local scale-invariant features," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, 1999, pp. 1150-1157.

[32] D. Luebke et al., "GPGPU: general-purpose computation on graphics hardware," in *SC '06 Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[33] A. Lumini, L. Nanni, and D. Maltoni, "Learning in Fingerprints," in *Biometrics: Theory, Methods, and Applications*, N. V. Boulgouris, K. N. Plataniotis, and E. Micheli-Tzanakou, Eds. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2009, ch. 14, pp. 339-364.

[34] D. Maio, D. Maltoni, R. Cappelli, J.L. Wayman, and A.K. Jain, "FVC2000: Fingerprint verification competition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 402-412, 2002.

[35] D. Maio, D. Maltoni, R. Cappelli, J.L. Wayman, and A.K. Jain, "FVC2002: Second fingerprint verification competition," in *International Conference on Pattern*

Recognition, vol. 16, 2002, pp. 811-814.

[36] D. Maio, D. Maltoni, R. Cappelli, J. L. Wayman, and A. K. Jain, "FVC2004: Third Fingerprint Verification Competition," in *proceedings International Conference on Biometric Authentication (ICBA04)*, Hong Kong, 2004, pp. 1-7.

[37] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*, 2nd ed.: Springer-Verlag New York, NJ, USA, 2009.

[38] P. Micikevicius. (2014, October) CUDA Optimization: Identifying Performance Limiters. [Online]. http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_dentifying_performance_limiters.pdf

[39] Luca Mussi, Fabio Daolio, and Stefano Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture," *Information Sciences*, vol. 181, no. 20, pp. 4642-4657, October 2011.

[40] K. Nandakumar, "Fingerprint matching based on minutiae phase spectrum," in *5th IAPR International Conference on Biometrics (ICB)*, 2012, pp. 216-221.

[41] C. Nugteren, G.J. van den Braak, H. Corporaal, and B. Mesman, "High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs," in *Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4*, New York, 2011, pp. 1-8.

[42] NVIDIA. (2014, October) CUDA C Best Practices Guide. [Online]. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[43] NVIDIA. (2014, October) CUDA C Programming Guide. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[44] NVIDIA. (2014, October) NVIDIA: Fermi compute architecture. [Online]. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[45] NVIDIA. (2014, October) Visual Profiler. [Online]. https://developer.nvidia.com/nvidia-visual-profiler

[46] D. Peralta, I. Triguero, R. Sanchez-Reillo, F. Herrera, and J.M. Benitez, "Fast fingerprint identification for large databases," *Pattern Recognition*, vol. 47, no. 2, pp. 588-602, February 2014.

[47] N.K. Ratha, K. Karu, S. Chen, and A.K. Jain, "A real-time matching system for large fingerprint databases," *IEEE Transactions on Pattern Analalysis and Machine Intelligence*, vol. 18, no. 8, pp. 799-813, August 1996.

[48] R. Shams and R.A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, Gold Coast, Australia, 2007, pp. 418–422.

[49] Weiguo Sheng, G. Howells, Michael Fairhurst, and F. Deravi, "A Memetic Fingerprint Matching Algorithm," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 3, pp. 402-412, September 2007.

[50] W. Sheng, G. Howells, M.C. Fairhurst, F. Deravi, and K. Harmer, "Consensus fingerprint matching with genetically optimised approach," *Pattern Recognition*, vol. 42, no. 7, pp. 1399-1407, July 2009.

[51] Unique Identification Authority of India, "Role of Biometric Technology in Aadhaar Enrollment," 2012.

[52] Universidad Autonoma de Madrid. (2013, November) Biometric Recognition Group - ATVS web site. [Online]. http://atvs.ii.uam.es/

[53] B. Yang, D. Hartung, K. Simoens, and C. Busch, "Dynamic Random Projection for Biometric Template Protection," in *in proocedings Fourth IEEE International Conference on Biometrics: Theory Applications and Systems (BTAS)*, 2010, pp. 1-7.

[54] Yin Zhang, Deng Yi, Baogang Wei, and Yueting Zhuang, "A GPU-accelerated non-negative sparse latent semantic analysis algorithm for social tagging data," *Information Sciences*, May 2014, [Available online].