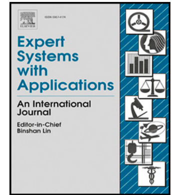
Expert Systems With Applications 224 (2023) 119955

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



Fast parallel computation of reduced row echelon form to find the minimum distance of linear codes

Manuel P. Cuéllar [a], F.J. Lobillo [b,c], Gabriel Navarro [a,c,*]

[a] Department of Computer Science and Artificial Intelligence, University of Granada, Calle Periodista Daniel Saucedo Aranda s/n, E18071, Granada, Spain
[b] Department of Algebra, University of Granada, Calle Fuente Nueva s/n, E18071, Granada, Spain
[c] Centro de Investigación en Tecnologías de la Información y de las Comunicaciones (CITIC), University of Granada, Calle Periodista Rafael Gómez Montero 2, E18071, Granada, Spain

ARTICLE INFO

Keywords:
Linear code
Minimum distance
Genetic algorithm
Parallel genetic algorithm
GPU-based parallel model
Post-quantum cryptography

ABSTRACT

Finding the distance of linear codes is a key aspect to build error correcting codes, and also to design attacks in code-based post-quantum cryptography; however, it is a NP-hard problem difficult to be addressed. Metaheuristics, and more specifically genetic algorithms, have proven to be a promising tool to improve the search of an upper bound for the distance of a given linear code. In a previous work, it was demonstrated that the there exists a column permutation of a code matrix whose Reduced Row Echelon Form (RREF) contains a row of minimum weight, i.e. the code distance, although calculating RREF during fitness evaluation increases the time complexity of the algorithm substantially. In this work, we propose parallelization of multiple calculations of Reduced Row Echelon Forms simultaneously, and its integration into a fully parallelized design of a CHC evolutionary algorithm to overcome this limitation. Moreover, we demonstrate empirically a substantial improvement in time complexity for the approach in practical case studies to find the distance of linear codes over different finite fields.

1. Introduction

Error correcting codes (Huffmann & Pless, 2003) are used to encode data sent from a source to a destination over a noisy channel, so that the receiver can recover the original message even if it was altered during the transfer due to the noise. Assuming that the original message is composed of a sequence of k symbols from an alphabet, this is achieved by adding r additional redundancy symbols to the message. This process is called *encoding*, and the resulting string of length $n = k + r$ is called a *codeword*. The encoding process has been traditionally tackled by endowing the code with some algebraic structure and selecting a finite field $\mathbb{F}$ as the alphabet. Among all error correcting codes, the family of linear codes are some of the most popular techniques to perform this task, and they have been successfully implemented in DVD/Blu-Ray discs, deep space transmissions, or wireless communications just to mention a few applications (Huffmann & Pless, 2003).

A linear code C is noted as $[n, k]_q$, and it is able to encode up to q^k messages of length k symbols into codewords of length n. It is a k-dimensional vector subspace of $\mathbb{F}_q^n$, where q is the number of elements in the finite field $\mathbb{F}_q$. A linear code is specified by a generating matrix $G_{k,n}$, and the encoding process of a message x of length k provides a codeword c of length n as the product $c = xG$. Its error correction capability depends on the code distance $d(C)$, which is calculated as the minimum weight of all codewords $d(C) = min\{w(c), c \in C\}$, where $w(c)$ is the number of nonzero elements in the codeword c (i.e. the Hamming distance of c to the vector 0 of length n). The error correction capability of a linear code is also bounded by the perfect error correction theorem of Richard Hamming (Hamming, 1950), so that to detect and correct an error in e symbols of a message, then the code distance must fulfill $d(C) \geq 2e + 1$.

Despite the simplicity of the definition, calculating a word of a linear code with length $d(C)$ is a NP-hard problem (Vardy, 1997), and so it is to know the right error correction capability of the code. To overcome this limitation, traditional approaches attempt to impose additional constraints over linear codes to ensure a lower bound for $d(C)$. As a result, subfamilies of linear codes such as BCH, Goppa, Reed–Solomon, Reed–Muller, etc. Huffmann and Pless (2003) are proposed in the literature. In these cases, having a lower bound for $d(C)$ ensures a minimum error correction capability, although the true error correction capability remains unknown. Although these alternatives can palliate the problem partially for error correction applications, finding the exact

* Corresponding author at: Department of Computer Science and Artificial Intelligence, University of Granada, Calle Periodista Daniel Saucedo Aranda s/n, E18071, Granada, Spain.
E-mail addresses: manupc@decsai.ugr.es (M.P. Cuéllar), jlobillo@ugr.es (F.J. Lobillo), gnavarro@ugr.es (G. Navarro).

https://doi.org/10.1016/j.eswa.2023.119955
Received 14 May 2021; Received in revised form 13 March 2023; Accepted 20 March 2023
Available online 23 March 2023
0957-4174/© 2023 Elsevier Ltd. All rights reserved.

$d(\mathcal{C})$ is also of interest in code-based post-quantum cryptography (Overbeck & Sendrier, 2009). Here, the secret key is a code from a family of codes for which an efficient decoding algorithm is known, that is masked so that it looks like a random linear code. In these cases, knowing a codeword of minimum distance may lead to the design of attacks to this type of codes.

There are different approaches in the literature to tackle the problem of calculating $d(\mathcal{C})$, by setting the problem forth as an optimization task. The most known is the Brouwer and Zimmermann's algorithm (Betten et al., 2006) and its extension in Lisonek and Trummer (2016), which is an exact algorithm that performs heuristic search over the solution space. In the worst case, the complete solution space has to be explored, although it can be used for small codes with a relatively affordable execution time. Other algorithms attempt to approximate lower/upper bounds for the code distance using approximate algorithms with probability estimations (Leon, 1988). Metaheuristics have also been used to find an upper bound of $d(\mathcal{C})$, as for instance Simulated Annealing (Muxiang & Fulong, 1994), Tabu Search and Ant Colony Optimization (Bland, 2007), although Genetic Algorithms have been more studied for both generic linear codes (Barbieri et al., 2005; Dontas & De Jong, 1990) and specific subfamilies such as QR codes (Nouh & Belkasmi, 2013), or BCH and DCC codes (Askali et al., 2013).

Most of the aforementined metaheuristics methods attempt to find a linear combination x of rows of generating matrix G that minimizes the weight $w(xG)$. In the recent work (Cuéllar et al., 2021) it was demonstrated that there exists a column permutation of the generating matrix whose Reduced Row Echelon Form (RREF) (Liesen & Mehrmann, 2015) contains a row r so that $w(r) = d(\mathcal{C})$. The main outcome of this method allows to change the traditional order solution representation $x \in \mathbb{F}_q^k$ to a permutation representation $x \in \mathcal{S}_n$, where $\mathcal{S}_n$ is the set of permutations of n symbols. The authors used this result to design genetic algorithms to perform a search over the space $\mathcal{S}_n$ to provide a column permutation of G whose RREF contains a row with optimal $d(\mathcal{C})$. This approach shows good results in practice and it is able to overcome local optima of previous approaches. However, the main limitation lies in the computation cost of the RREF method for a solution evaluation.

In this work, we deepen into the approach of Cuéllar et al. (2021), and provide an adaptation of evolutionary algorithms and fitness calculation to high-performance computing models. Concretely, we propose an adaptation of the evolutionary CHC algorithm described in Cuéllar et al. (2021) and RREF calculation for its parallel implementation in high-performance general-purpose computing on graphics processing units (GPGPU). We study different models of parallelization, and provide experimental results to test each approach. Moreover, we also study the special case where the underlying finite field is $GF(2)$, which is of special interest due to its particular use in real applications.

The manuscript is structured as follows: Section 2 describes the existing related works in the literature, for parallelization of both evolutionary algorithms and RREF calculation. Then, Section 3 describes the parallel computing model, the design of parallel components of the algorithm and solution evaluation. Section 4 shows a performance analysis of the designed parallel components, and applications to real scenarios. Finally Section 5 concludes and provides future works.

2. Related work

We may highlight two main trends in the design of contemporary distributed and/or parallel algorithms for high-performance computing systems. One follows data-oriented design principles, as for instance the former Map-Reduce from Google or the Resilient Distributed Dataset of Apache Spark paradigms (Stephen et al., 2017), both widely used in Big Data applications. The models in this category attempt to divide large amounts of data among different computing units that run the same algorithm over splits of the dataset. Then, the partial outcomes are aggregated to obtain the final result, similarly to a *Divide-and-Conquer* approach. The models within the second trend attempt to split algorithms themselves into small components that run in separate computing units in parallel, each one in charge of running specific code and interacting with the other components to build the desired algorithm behavior. These models are of special interest when the dataset is not as large as in a Big Data scenario, but the computational cost of the algorithm is high. Here, we can also distinguish between pure parallel algorithm designs, or parallelization of some components of the algorithm. Let us focus on the case of evolutionary algorithms, which is a central part of this manuscript. In the former case, we cite as example the classic Parallel Genetic Algorithm (Mühlenbein & Born, 1991), the Master–Slave (Cantú-Paz, 1998) or the island model of Genetic Algorithms (Cantú-Paz, 1998). On the other hand, the latter case has been studied in the last decade and, in the case of genetic algorithms, we can find multiple full algorithm component parallelizations, as for instance (Pospichal et al., 2010).

The proposal of this manuscript clearly lies within the second trend, since our dataset is a single generating matrix $G_{k,n}$ which, in a worst scenario, requires a size of a few MB of computer memory. However, the estimated computational cost is high since the execution of a genetic algorithm (GA) requires the evaluation of multiple solutions at each iteration. Moreover, as it was mentioned in the introduction, our approach is based in the results of Cuéllar et al. (2021), where evolutionary algorithms were used to evolve populations of permutations in $\mathcal{S}_n$, and the fitness function requires the calculation of the RREF, with complexity in $\mathcal{O}(kn^2)$. To solve these limitations, we propose a full parallelization of the components of the algorithm and fitness calculation.

As it has been mentioned, there has been a big effort to propose parallel models of evolutionary algorithms and their implementation in GPGPU since the last decade. Examples are the implementation of island models and master–slave strategies to solve the problem of temporal dynamics of gene regulatory networks (García-Calvo et al., 2018), the implementation of island models in Nesmachnow et al. (2010) to solve scheduling problems using heterogeneous computing and grid systems and flexible flow shop scheduling problems in Luo et al. (2018), or a general implementation of the master–slave model for the Python language in Skorpil et al. (2019).

The literature also offers papers describing full parallelization of the components of an evolutionary algorithm, as for instance the works (Pospichal et al., 2010; Sinha et al., 2016; Luo et al., 2019; Berisha et al., 2017; Xie & Ning, 2013; Chen et al., 2011). These approaches share the same basic parallelization principles of components of GAs and their implementation in GPGPUs; however, each paper addresses a different problem with different solution representation, crossover, mutation and fitness evaluation. In our case, we propose a full parallelization of the CHC algorithm described in Cuéllar et al. (2021). Other authors have also tackled the problem of creating a parallel version of CHC, as in Bilbao and Leguizamón (2019). Here, the authors propose the CHC to find optimal locations for aerogenerators in wind farms, and propose an island model to distribute different solution populations over multiple computing cores. The closest paper to the approach described in this manuscript that we have found in the literature is (Filipiak & Lipiński, 2012), which proposes a parallelization of the CHC algorithm to solve the traveling salesman problem. Here, the recombination and evaluation are performed in parallel, and also a new operator for local search, included to improve solution quality. Our approach goes beyond the ideas of Filipiak and Lipiński (2012), and we propose to parallelize the whole evolutionary process including initialization, parent selection, recombination threshold calculation and replacement operators of the classic CHC algorithm (Eshelman, 1991).

To conclude with the description of the related work regarding GA parallelization, we cite the survey in Umbarkar and Joshi (2013), which provides a taxonomy of parallelization models of evolutionary algorithms to help the interested reader to deepen into specific parallelization techniques as an initial literature search point.

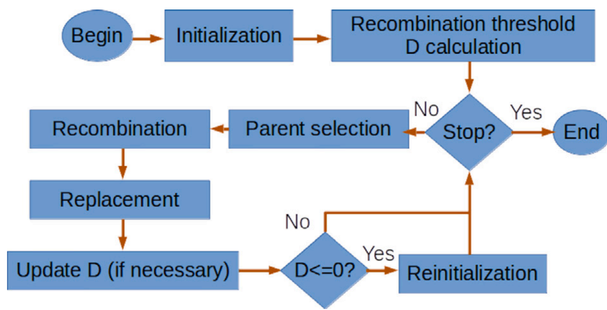


Fig. 1. Diagram of the main computation flow of CHC.

On the other hand, there is not much literature regarding the parallel calculation of RREF, although we may cite the papers (Dumas et al., 2014; Linton et al., 2018). In Dumas et al. (2014), five different proposals are studied for Gauss elimination over finite fields using OpenMP technology. Of special interest to our work is the approach to find Gauss elimination with no rank defficiency, as the generating matrices of linear codes are not rank deficient. In this approach, computations are split into column tiles that are processed in parallel. Our work differs substantially from this paper, since our underlying parallel model is based on GPGPU architectures. Moreover, we target at multiple RREF calculations (one for each individual in the population), so that a different parallelization model is required rather than for calculation of a single RREF. Similarly, the work in Linton et al. (2018) attempts to perform Gaussian elimination by dividing a matrix into square submatrices, and takes advantages of concurrency to process each individual block in parallel on different computing units. The paper assumes that the objective is also to calculate Gauss elimination over a single matrix, and takes advantage of all computing units to propose the parallel algorithm. In our work, we propose a different parallel model that can be used to calculate multiple RREFs in parallel and, in contrast to previous approaches, it can be integrated easily into a genetic algorithm computing flow.

3. Parallel model and design

In this section, we first review the CHC algorithm and its components to make the article self-contained. The goal is not to provide a full description of the algorithm, which can be found in Eshelman (1991), Cuéllar et al. (2021), but to outline its main behavior so that we may distinguish candidate components to be parallelized. After that, Section 3.1 describes the ground model used to build the parallel algorithm, whose components are described in Sections 3.2–3.7.

The CHC algorithm is an evolutionary algorithm that holds a balance between diversity and convergence, where diversity focus on genotypic distances among solutions in the population. It is based on four main components: elitist selection, the HUX recombination operator, an incest prevention check to avoid the recombination of similar solutions, and a population reinitialization method when a local optimum is found. Fig. 1 shows the computing flow of the algorithm, whose description is as follows:

First a population P is initialized and evaluated according to a fitness measure. After that, a recombination threshold D is calculated as the average distance of individuals in P, being the Hamming distance the criterion used in this article. A recombination threshold decrement value is also computed as $dec = \tau D_{max}$, where D_{max} is the maximum distance among solutions in the population and τ is an hyperparameter, usually $\tau \in [0, 1]$. Then, the main loop of the algorithm starts by checking if a stopping criterion is satisfied (i.e. to have a number of solutions evaluated, to obtain a solution with a given fitness, etc.). While this criterion does not hold, the individuals in the population are shuffled and matched by pairs (parent selection). Each pair of parents are then combined using a crossover operator (recombination) to generate two new solutions, only if the distance between the parent solutions is over D. This provides a population of *children* containing between 0 and $|P|$ individuals, depending on how many parents recombined, and the fitness of the *children* is calculated. The best solutions among parents and children will replace P for the next algorithm's iteration (replacement). If either no child was generated during crossover, or the population at next iteration equals the population at current iteration, the recombination threshold is updated as $D \leftarrow D - dec$ and, if D is under or equals 0, then P is replaced with new random solutions and the best solution found until the current iteration (reinitialization with elitism).

In our work, our population is a set of permutations of n elements, and the fitness calculation requires the computation of the RREF of a generating matrix $G_{k,n}$ of a linear code whose columns are permuted according to each individual. Thus, the components of the algorithm that can be parallelized are the initialization/reinitialization of the population, the computation of the recombination threshold, the crossover, the replacement and the fitness calculation. The parent selection is calculated as a random permutation of $|P|$ elements, which can reuse the initialization algorithm to generate a single permutation of $|P|$ elements. Next, we describe the parallelization model and the parallel proposals.

3.1. The parallel model

Nowadays, the Compute Unified Device Architecture (CUDA) is the most used model for GPGPU programming among all parallelization models, due the NVIDIA® supremacy in the market. A generalization of this model is also adopted by the Open Computing Language (OpenCL), which is intended to be a standard. Although the CUDA architecture is well known and a deep description can be found in NVIDIA et al. (2020), the next paragraphs provide an outline for article self-completeness and to introduce the notation used in the next subsections.

There are two main concepts in CUDA parallelization: The *host* (usually a CPU), that controls the computing flow of the main program, and the *device*, which is a GPU or cluster of GPUs able to run parallel programs. Parallel programs (*kernels*) in CUDA run on a *grid*. When a *kernel* is launched, the *host* must indicate the number of blocks and threads per block used to executed the *kernel*, so that the *grid* contains a set of $|B|$ *blocks*, and each *block* contains the same number of *threads* $|T|$. Each block in the device grid is identified by an index B_i ranging from 0 to $|B|-1$, and each thread within a block is identified by a local thread index LT_j ranging from 0 to $|T|-1$. Thus, each thread in the device can be uniquely identified by the global index $GT_k = LT_j + |T|B_i$. CUDA is built upon the principle "Single Instruction Multiple Threads" (SIMT), so that a thread is the minimum parallel computation unit, and all threads must run the same *kernel* at the same time.

There is a hierarchy of working memory in CUDA, which essentially organizes memory in *Global* memory (slower, accessible by all threads in all blocks, massive -order of GB-, persistent during a CUDA session), *Shared* memory (faster than global memory, shared by all threads in a block, in the order of a few KB, volatile among different kernel executions), and *Local* memory (very fast, private to each thread, composed of a few registers or very few KB). All threads in the same block can share information through the shared memory and can synchronize execution during a *kernel* run, but threads among different blocks cannot share information nor synchronize execution.

Considering the previous constraints, the proposed parallel model consists of a *host* that runs the main computing flow of the CHC algorithm in Fig. 1, and each component of the algorithm is executed as a *kernel* on the *device*. We design each *kernel* so that each block is assigned with a sub-task that requires no interaction with other blocks, but threads in the same block may collaborate to build the desired *kernel* behavior for their corresponding sub-task. Since memory transfer

between *host* and *device* can be a bottleneck, our design assumes that all relevant information for kernels is stored in the *device* global memory during the CHC algorithm execution (i.e. population, fitness, offspring, distance between individuals). Algorithm 1 outlines the *host* program, where NC is the number of children generated during replacement and $NewC$ the number of children that are included in the population at the next algorithm iteration. The next sections describe each *kernel* in detail.

Algorithm 1: Computing flow control of CHC at *host*

$|P|$: Population size
τ : Recombination threshold decrement hyperparameter
$G_{k,n}$: Generating matrix of the code
1 Memory allocation in *device* for $2\cdot|P|$ solutions, their fitness, and distance
2 Call Initialization kernel
3 Call Evaluation kernel
4 Download best fitness and solution from *device* to *host*
5 Call Recombination threshold calculation kernel
6 Call Recombination threshold aggregation kernel
7 Download D, dec from *device* to *host*
8 **while** *stopping criterion not satisfied* **do**
9 | Call Parent Selection kernel
10 | Call Recombination kernel
11 | Download NC from *device* to *host*
12 | **if** NC=*0* **then**
13 | | Update $D \leftarrow D - dec$
14 | **else**
15 | | Call Evaluation kernel
16 | | Call Replacement kernel
17 | | Download $NewC$ from *device* to *host*
18 | | **if** $NewC$=*0* **then**
19 | | | Update $D \leftarrow D - dec$
20 | | **else**
21 | | | Download best fitness and solution from *device* to *host*
22 | **if** $D \leq 0$ **then**
23 | | Call Reinitialization kernel
24 | | Call Evaluation kernel
25 | | Download best fitness and solution from *device* to *host*
26 | | Call Recombination threshold calculation kernel
27 | | Call Recombination threshold aggregation kernel
28 | | Download D, dec from *device* to *host*

3.2. Initialization and reinitialization

The *Initialization/Reinitialization* kernel is in charge of generating a population of $|P|$ random permutations of n elements $P_0, P_1, \dots, P_{|P|-1}$ in device global memory. If *Reinitialization* takes place, then only $|P|-1$ solutions are generated randomly, and the best solution found during the evolutionary process is inserted in the population as the remaining permutation. Although initialization/reinitialization is not frequently executed during the CHC algorithm runcycle, and therefore is not a bottleneck, its parallelization might help to speed up execution. The proposed parallel initialization kernel is straightforward. Its design assumes that each thread GT_i in the device is in charge of generating solution P_i in the population, and runs the code described in Algorithm 2. The permutation P_i is created by exchanging a random position $P_i(k)$ and $P_i(j)$ for each component $j = 1..n$. The fitness of the corresponding solution is initialized to *Unknown* so that evaluation kernel can latter distinguish the solutions that require evaluation. An exception is made for thread with global index 0 which, in the case of reinitialization, performs a copy of the best solution found to the population individual P_0.

The time complexity of each kernel execution is $\mathcal{O}(n)$ if the number of simultaneous running threads $|T| > |P|$. Being compared with the sequential version running in $\mathcal{O}(|P|n)$, the benefits of this parallelization strategy will increase with the population size. Finally, running this kernel requires to optimize the number of blocks B and threads per block T to be instantiated by the host call, and this is an experimental problem that is solved in Section 5.

Algorithm 2: Initialization/Reinitialization kernel

P: Population in global memory
n: Permutation length (scalar)
reinit: flag with value *true* if reinitialization, or *false* if initialization
Best : best solution found (global memory)
Fitness: Fitness of solutions (global memory)
1 **if** *reinit and* $GT_i = 0$ **then**
2 | $Fitness(i)$ = Fitness of Best
3 | **foreach** *j=1..n* **do**
4 | | $P_i(j) = BestSol(j)$
5 **else**
6 | $Fitness(i)$ = Unknown
7 | **foreach** *j=1..n* **do**
8 | | $P_i(j) = j$
9 | **foreach** *j=1..n* **do**
10 | | k= random value in $1..n$
11 | | Exchange values of $P_i(j), P_i(k)$

3.3. Recombination threshold

The recombination threshold D is initialized to the average distance between all solutions in the population as $D = \frac{2}{|P|(|P|-1)} \sum_{i=0}^{|P|-1} D(i)$, where $D(i) = \sum_{l=i+1}^{|P|} d(i,l)$. In this work, $d(i,l) = \sum_{k=0}^{n-1} (P_i(k) \neq (P_l(k)))$ is the Hamming distance between P_i and P_l, where n is the permutation length and the operator $\neq$ returns 1 if $P_i(k)$ and $P_l(k)$ contain different values, and returns 0 otherwise. The time complexity to compute D with a sequential algorithm is $\mathcal{O}(n|P|^2)$.

The idea behind the parallelization of calculation of D is to separate the computation of each value $D(i)$, and then aggregate these values to obtain D. Thus, our parallel design matches block B_i in the grid with solution P_i, and each block is in charge of calculating $D(i)$. Every thread LT_j within a block calculates a portion of $d(i,l)$. More specifically, each thread computes the distance among components $P_i(j + k|T|)$ and $P_l(j + k|T|), \forall k : j + k|T| \leq n$, and update a shared memory variable denoted as D_i with these partial results. Algorithm 3 summarizes the parallelization of calculation of $D(i)$. To improve execution time, we also assume that P_i is in shared memory for each block B_i, respectively. Algorithm 3 runs in $\mathcal{O}(n|P|/|T|)$ and, if $|T| >= n$, it can achieve $\mathcal{O}(|P|)$. We remark that an *atomic update* in Algorithm 3 and later algorithms means operations that are executed as a single, undivisible computer instruction that cannot be splitted nor subdivided in other simpler operations. Atomic operations avoid concurrent interferences and unexpected behaviors when different threads write to the same memory space.

As no synchronization is allowed among blocks in the grid, another kernel is required to aggregate all values $D(i)$ when all blocks finish running Algorithm 3. Algorithm 4 is in charge of making this aggregation. As there must be information exchange between threads to make this aggregation, a single block must be used. We designed the aggregation using the reduction technique. Without loss of generality, we assume $|P|$ to be a power of 2 for the explanation, and a number of threads $k = |T| = |P|/2$ for optimal kernel execution. Each thread LT_i aggregates the values D_i and D_{i+k}. Then, the process is repeated updating $k = \lfloor k/2 \rfloor$ until $k = 0$, and thread 0 is responsible to average the aggregated values. Before each new iteration begins, the threads are synchronized to ensure that all values D_i have been updated. Fig. 2 shows an example of aggregation of an array of D_i values for a population of $|P| = 8$ individuals. As we may see, the algorithm runs with a time complexity of $\mathcal{O}(log_2(|P|))$.

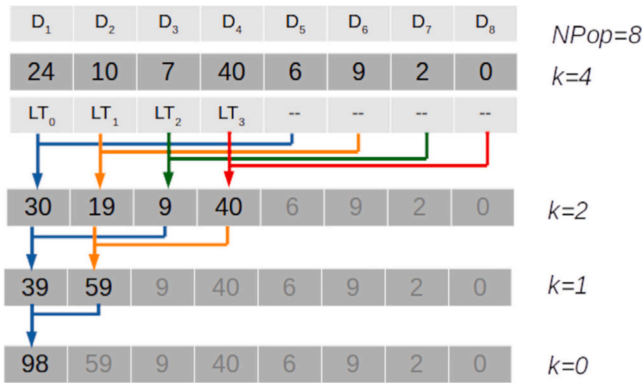


Fig. 2. Example of recombination threshold aggregation kernel with population size $|P|$=8.

Algorithm 3: Recombination threshold calculation kernel

P: Population in global memory
D_i**:** Sum of distances D(i) of P_i to $P_l, \forall l > i$
Data: LD_i: Shared memory distance calculated by threads in block B_i

```
1  if LT_j = 0 then
2  |  LD_i = 0
3  synchronize threads
4  distance= 0
5  for l=B_i + 1 to |P| do
6  |  for k=LT_j to n in steps of |T| do
7  |  |  Update distance= distance+d(P_i(k), P_l(k))
8  Update atomically shared value LD_i = LD_i+distance
9  synchronize threads
10 if LT_i = 0 then
11 |  D_i = LD_i
```

Algorithm 4: Recombination threshold aggregation kernel

$(D_0, D_1, ..., D_{|P|-1})$**:** Distances $D(i)$ calculated by Algorithm 3
D : Average distance of all members of the population

```
1  k = |T|
2  while k ≥ 1 do
3  |  if LT_i < k then
4  |  |  Update D_i = D_{i+k}
5  |  Update k = ⌊k/2⌋
6  |  Synchronize threads
7  if LT_i = 0 then
8  |  D = D_0 2/(|P|(|P| − 1))
```

3.4. Parent selection

As we mentioned in Section 3, parent selection is a special case of the initialization kernel since a single permutation of $|P|$ values has to be generated. We name this permutation as *parents*. Solutions in the population $P_{parents[2i]}$ and $P_{parents[2i+1]}$, $0 \leq i < |P|/2$, are matched to generate new solutions using the crossover procedure. Thus, the initialization kernel is reused to perform parent selection whose execution requires one block and one thread only. It is equivalent to a sequential procedure, and therefore the time complexity of the kernel call is $\mathcal{O}(|P|)$.

3.5. Crossover

We use the algebraic crossover operator AX_2 described in Cuéllar et al. (2021), under the hypothesis that the composition of permutations with good fitness, may produce a solution with better fitness. Given two parents P_1 and P_2 to be recombined, two new children are generated as $C_1 = P_1 \circ P_2$, $C_2 = P_2 \circ P_1$, where $\circ$ stands for the permutation composition operator.

Algorithm 5: Recombination kernel

P: Population
parents: indices of parents to be combined
C : Generated children
Fitness : Updated fitness
Data: LD_i, shared memory containing the distance between the i-th pair of parents

```
1  if LT_j = 0 then
2  |  LD_i = 0
3  localDistance= 0
4  for k=LT_j to n in steps of |T| do
5  |  update localDistance=
   |    localDistance+d(P_parents[2i](k), P_parents[2i+1](k))
6  update atomically LD_i = LD_i+localDistance
7  Synchronize threads
8  if LD_i ≥ D then
9  |  if LT_j = 0 then
10 |  |  fitness(2i) = Unknown
11 |  |  fitness(2i + 1) = Unknown
12 |  for k=LT_j to n in steps of |T| do
13 |  |  C_2i(k) = P_parents[2i](k)∘P_parents[2i+1](k)
14 |  |  C_2i+1(k) = P_parents[2i+1](k)∘P_parents[2i](k)
15 else
16 |  if LT_j = 0 then
17 |  |  fitness(2i) = None;
18 |  |  fitness(2i + 1) = None;
```

As the composition of two permutations of size n can be implemented in $\mathcal{O}(n)$, and we need to generate $|P|$ compositions in the worst case, the sequential version of the crossover can be implemented in $\mathcal{O}(n|P|)$. The proposed parallel kernel is designed to run $|P|/2$ blocks with n threads per block. Then the crossover for the entire population can be performed in $\mathcal{O}(|P|n/(|T||B|))$. If $|B| \geq |P|$ and $|T| \geq n$, then the time complexity of the method is $\mathcal{O}(1)$. In our kernel, a block B_i is in charge of the recombination of $P_{parents[2i]}$, $P_{parents[2i+1]}$ to generate children C_{2i}, C_{2i+1}, and each thread within the block is in charge of combining a portion of approximately $\lceil n/|T| \rceil$ components of the parents.

The proposed kernel also calculates incest prevention, i.e. two parents P_i, P_j are combined if $d(i,j) \geq D$. Thus, the kernel is divided into two parts: First, the distance between parents is calculated. If this distance is over the recombination threshold D, then recombination takes place for those parents, where each block's thread LT_j recombines components $LT_j + k|T|, \forall k : j + k|T| \leq n$ from the source parents. In this case, the fitness of the generated children is tagged as *Unknown* so that the evaluation kernel can distinguish which solutions must be evaluated. Otherwise, the children's fitness is tagged as *None* to indicate that the children does not exists. Algorithm 5 outlines the kernel run by each thread.

3.6. Replacement

The replacement operator generates population P at next iteration considering the best $|P|$ solutions among population and children at the current iteration. The replacement is easily performed by creating a joint population $P' = P \bigcup C$ of maximum size $2|P|$, and indexing each individual in P' using indices $idx[0..2|P|-1]$. Then, a sorting procedure

is performed with the criterion $idx[i] < idx[j] \leftrightarrow fitness[i] < fitness[j]$, and the individuals whose sorted indices are in $idx[0]..idx[|P|-1]$ will become population at the next iteration. The best way to do this with a sequential algorithm performs in $\mathcal{O}(|P|log_2(|P|))$ using a fast sorting method. The same time complexity can be achieved in the parallel replacement kernel proposed in this work, although a practical improvement in performance can be obtained.

Any parallel sorting algorithm could be used to fulfill the purpose of replacement, as for instance (Casanova et al., 2017). However, current sorting methods designed for GPU attempt to sort very large arrays, which is not the case in our work. Our population, in the worst case, is expected to have a very few thousands of individuals, so that complex parallel sorting strategies are not required. In our case, a basic approach is suitable for our purposes as in Davidson et al. (2012).

The kernel is designed using the reduction technique, and it is based on the *merge* operation of the well-known algorithm *MergeSort*. As there must be communication between all threads to perform parallel sorting, the kernel must be executed in a single block. Without loss of generality, we assume that $|P|$ is power of 2, and the number of threads to call the kernel is $|T| = |P|/2$ for explanation purposes.

The kernel works as follows: First, at iteration $it = 1$ a number of $k = |T| = |P|/2$ threads combine consecutive lists of $2^{it-1} = 1$ elements to provide a sorted list of 2 elements. Next, at iteration $it = 2$ we update $k = k/2$ and each thread is assigned with two consecutive lists of $2^{it-1} = 2$ sorted elements, which are merged into a new list of 4 sorted elements. The process repeats until a single list of $|P|$ elements is obtained. Algorithm 6 outlines the kernel algorithm run by each thread, and Fig. 3 shows an example of sorting a population of $2|P| = 8$ individuals and 4 threads.

Algorithm 6: Replacement kernel

$P \bigcup C$: Joint population of parents and children
idx : array containing the index of each individual in $P \bigcup C$
fitness: array containing the fitness of individuals in $P \bigcup C$
Data: *out*, local memory array to store partial merged lists

```
1  it= 0
2  while 2^{it-1} < |P| do
3      begin = min{2|P|, 2LT_i 2^{it-1}}
4      end = min{2|P|, 2LT_{i+1} 2^{it-1}}
5      center = (begin + end)/2
6      i = begin, j = center
7      k = begin
8      while i < center and j < end do
9          if fitness[idx[i]] <= fitness[idx[j]] then
10             out[k]= idx[i]
11             update i=i+1, k=k+1
12         else
13             out[k]= idx[j]
14             update j=j+1, k=k+1
15     while i < center do
16         out[k]= idx[i]
17         update i=i+1, k=k+1
18     while j < end do
19         out[k]= idx[j]
20         update j=j+1, k=k+1
21     update it= it+1
22     Synchronize threads
23     foreach k in begin..end do
24         idx[k]= out[k]
25     Synchronize threads
```

3.7. Fitness calculation

Let $G_{k,n}$ be a generating matrix for a linear code C, $x \in \mathcal{S}_n$ a permutation of n columns of $G_{k,n}$, and $G' = RREF(G,x)$ a procedure

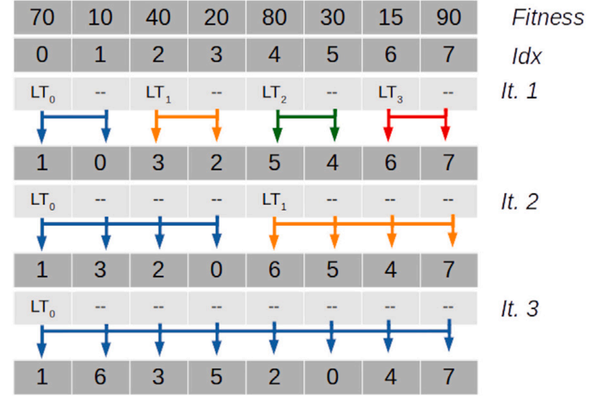


Fig. 3. Example of replacement kernel with population size $2|P| = 8$.

that calculates G', the Reduced Row Echelon Form of matrix G whose columns are permuted according to x. Theorem 1 in Cuéllar et al. (2021) states that there exists a permutation $x^* \in \mathcal{S}_n$ of n columns of $G_{k,n}$ for which a row r of the matrix $G' = RREF(G, x^*)$ fulfills $w(r) = d(C)$. Let us rewrite G' as a list of rows of length n, $G' = \{r_1, r_2, \dots, r_k\}$. The fitness $f(x)$ of a solution x in the population is then calculated as it is shown in Eq. (1), and the algorithm must find the solution $x^* = argmin_x\{f(x)\}$.

$$f(x) = min\{w(r_i), \forall r_i \in G'\} \tag{1}$$

The traditional method for computing the RREF has a time complexity in $\mathcal{O}(kn^2)$ (Liesen & Mehrmann, 2015). If the population contains $|P|$ solutions whose fitness must be evaluated, then the evaluation sequential procedure has a time complexity of $\mathcal{O}(|P|kn^2)$.

We propose a parallelization of the computation of RREF to reduce the time complexity of population evaluation. The proposed kernel behaves in time complexity $\mathcal{O}(n\lceil k/|T|\rceil)$, considering it can run in $|P|$ blocks of $|T|$ threads per block. If $k = |T|$, then the time complexity is $\mathcal{O}(n)$. The design of the kernel is as follows: Each block B_i is in charge of evaluating a solution P_i in the population, and each thread LT_j in a block is responsible to perform the required operations to calculate row j of the resulting matrix G'. Algorithm 7 outlines the basic kernel procedure. First, each thread LT_j makes a local copy of row j of matrix G, whose components are permuted according to solution P_{B_i} in the population, only if the solution requires evaluation. After that, the algorithm starts traveling rows from $cRow = 0$ and columns from $cCol = 0$ and, for each row, it calculates the pivot (first row in current column with a non-zero value), storing the pivot row index in $idxPivot$. If no pivot exists for the current column, then the next column of the matrix is checked. Otherwise, the pivot located at column $cCol$ must be the unity value, and the thread in charge of $r_j = r_{idxPivot}$ updates $r_j = (r_j(cCol))^{-1} \cdot r_j$. After that, it is ensured that current row r_{cRow} contains the pivot, and finally, all remaining threads responsible of rows r_j where $r_j(cCol) \neq 0$ are updated to make value 0 in the current column. Thus, current column has all elements equal to 0 but the pivot row, with value 1. Next row and column are checked to continue the RREF calculation. Once the complete RREF has been calculated each thread LT_j calculates $w(r_j)$ and updates the fitness that corresponds to their block if it is necessary.

We remark that addition, product and inverse calculation in $\mathbb{F}_q$ have a time complexity of $\mathcal{O}(1)$. A special case is the finite field $\mathbb{F}_2$, where $i^{-1} = i$, the product can be implemented as logic *and* gates, and addition as logic *xor* gates. Also, binary codes are the most extended in the literature and the ones with most of practical applications. For this reason, a binary implementation of Algorithm 7 has been also included in the experiments, which uses bit-wise representation of matrix G in 32-bit words, and performs finite field operations as bit-wise logic

Algorithm 7: Evaluation kernel

```
P: Population to be evaluated
Fitness: array containing the fitness of individuals in P
Data: r_j, array with j-th row of G' to be calculated by the
      thread (local thread memory)
Data: rPivot, array containing the pivot row (shared memory)
Data: cRow, cCol, current row and column indices (shared
      memory)
Data: idxPivot, current pivot row (shared memory)
1  if fitness(B_i) is Unknown then
2    foreach k in 1..n do
3      r_j(P_{B_i}(k)) = G(j, k)
4    if LT_j = 0 then
5      fitness(i) = ∞
6      cRow, cCol= 0
7      idxPivot= k
8    Synchronize threads
9    while cRow < k and cCol < n do
10     if r_j(cCol) ≠ 0 then
11       Update atomically idxPivot= min{idxPivot, j}
12     Synchronize threads
13     if idxPivot=k then
14       cCol= cCol+1
15     else
16       if LT_j = idxPivot then
17         Update r_j = (r_j(cCol))^{-1} · r_j
18         copy r_j to shared rPivot
19       Synchronize threads
20       if LT_j = cRow and cRow ≠ idxPivot then
21         Update r_j = r_j + rPivot
22         Exchange r_j, rPivot
23       Synchronize threads
24       Update r_j = r_j + rPivot · r_j(cCol) if k ≠ cRow
25       Synchronize threads
26       if LT_j = 0 then
27         Update
           cRow = cRow + 1, cCol = cCol + 1, idxPivot = k
28       Synchronize threads
29   Calculate w(r_j)
30   Update atomically fitness(i) = min{fitness(i), w(r_j)}
```

operations. This implementation help to reduce the operations of RREF calculation in a ratio of $\lceil n/32 \rceil$ which, for contemporary codes, is a great advantage.

4. Experiments

The experiments are organized in two blocks: Section 4.1 describes an experimental study of the computational cost of parallel components of the CHC algorithm described in Section 3. Then, in Section 4.2 the complete CHC algorithm is applied over real datasets to study the computational cost in both $\mathbb{F}_8$ and $\mathbb{F}_2$ finite fields.

4.1. Performance study of parallel components

Here we study the computational cost of the parallel components described in Section 3. To do so, we design a set of experiments and calculate the computational cost of each component. The population size and the permutation length of individuals are varied in increasing size, denoted as a tuple $(|P|, n)$.

Also, different grid sizes are tested for each case, also described as a tuple *(Number of blocks, Threads per block)*. A special case of a grid size $(1,1)$ is included in all experiments, to study the gain of parallelization against a pure sequential algorithm, where a single block containing a single thread must perform all operations to achieve the desired results. The values $|B|$ and $|T|$ for the grid size vary for each experiment, and are set to the values we suggested during kernel description in Section 3. In addition, other extreme grid cases $(|B|, 1)$ and $(1, |T|)$ are studied for those components where it is possible to configure more than one block, as well as the suggested block size and threads per block for each kernel, i.e. $(|B|, |T|)$.

To be able to perform these comparisons, the kernels described in Section 3 were adapted so that each thread can accomplish the workload of multiple threads, and each block can accomplish the workload of multiple blocks. These kernels were implemented in CUDA, and we used Python language and PyCUDA library as control host of algorithm computing flow. All experiments were performed using a single NVIDIA GPU model GTX 1050 with 640 cores running at 1455 GHz, 2 GB RAM, 6 Streaming Multiprocessors, 1024 maximum threads per block.

Statistical tests were performed to know if there are significant differences among the different grid sizes for the same settings. To do so, we run 50 experiments for each population and grid sizes. Then, a normality Saphiro-Wilk test with confidence 95% was applied to know if all data distributions are drawn from a normal distribution. If so, then the results of the grid sizes were sorted in ascending order of average time (the median value was used instead of the mean if any data distribution is not parametric). A paired t-test was applied to compare distributions if all of them are parametric, and a Wilcoxon test was used if not, both with 95% confidence level.

Tables 1 to 7 show the results obtained to measure execution time of the proposed kernels. Rows contain the population sizes tested, and columns the grid sizes. Cells describe the average computational time spent by each setting, in seconds. In addition, subscripts are used to rank in ascending order the execution time according to the statistical tests. Value (1) is for the solutions that spent the lower computational time. If statistical tests concluded that there are no significant differences between two settings, then the same rank value (i) is plotted in the corresponding row cells. Population settings of *($|P|$, n)* equals to $(30, 50)$, $(100, 500)$, $(1000, 1500)$ and $(2000, 5000)$ were used for all components except for evaluation, where populations larger that $|P| = 100$ consumed a high computational time in sequential algorithms.

Regarding the initialization kernel, statistical tests concluded that using a grid of $|P|$ blocks with a single thread per block is the best choice for small population and permutation sizes, and there are significant differences with the second best choice. However, for a large population and permutation length, using the maximum threads per block available, and the minimum number of blocks, provide the best results. In this case, the parallel proposal achieves a time that is 102,31 times faster than the sequential version of the algorithm, and it is able to reduce the execution time from the order of seconds to the order of centiseconds.

If we consider the recombination threshold calculation kernel, Table 2 shows the experiments performed for increasing sizes of population and permutation length, with different grid configurations. We observe that the execution time of the sequential algorithm increases substantially and, for population sizes of $|P| = 1000$ with $n = 1500$, it exceeds tens of seconds. For larger populations, it can also exceed the time in minutes. However, the parallel kernel is able to reduce the time execution substantially, and we can obtain a gain ranging from 31,41 times faster (population size $(30, 50)$) to 786,58 times faster (population size of $(2000, 5000)$). The optimal grid setting in this case is to use a number of blocks equals to the population size, and a number of threads per block equals to the permutation length. Thus, multiple threads collaborate to calculate partially the distance of a solution matched with their corresponding block to the remaining solutions in the population. Even if the recombination threshold calculation is performed only after a few iterations, the time gain of the parallel kernel provides a huge advantage against the sequential algorithm, and may help to reduce the overall execution time of the algorithm.

Table 1
Computational time of initialization/reinitialization kernel.

Population	(1, 1)	(|P|, 1)	(1, min{1024, n})	(|P|/min{1024, n}, min{1024, n})
(30, 50)	$0.000830_{(4)}$	$0.000039_{(1)}$	$0.000044_{(3)}$	$0.000044_{(2)}$
(100, 500)	$0.019984_{(3)}$	$0.000298_{(1)}$	$0.000343_{(2)}$	$0.000344_{(2)}$
(1000, 1500)	$0.595605_{(4)}$	$0.005867_{(1)}$	$0.007643_{(3)}$	$0.007623_{(2)}$
(2000, 5000)	$3.974139_{(4)}$	$0.049915_{(2)}$	$0.052351_{(3)}$	$0.038843_{(1)}$

Table 2
Computational time of recombination threshold computation kernel.

Population	(1, 1)	(|P|, 1)	(1, min{1024, n})	(|P|, min{1024, n})
(30, 50)	$0.001068_{(4)}$	$0.000077_{(2)}$	$0.000416_{(3)}$	$0.000034_{(1)}$
(100, 500)	$0.072788_{(4)}$	$0.001525_{(2)}$	$0.004968_{(3)}$	$0.000335_{(1)}$
(1000, 1500)	$24.758136_{(4)}$	$0.207915_{(2)}$	$0.599276_{(3)}$	$0.073151_{(1)}$
(2000, 5000)	$343.210720_{(4)}$	$7.639736_{(3)}$	$3.283687_{(2)}$	$0.436334_{(1)}$

Table 3
Computational time of recombination threshold aggregation kernel.

Population	(1, 1)	(1, |P|)
(30, 50)	$0.000013_{(1)}$	$0.000067_{(2)}$
(100, 500)	$0.000014_{(1)}$	$0.000014_{(1)}$
(1000, 1500)	$0.000020_{(1)}$	$0.000020_{(1)}$
(2000, 5000)	$0.000036_{(1)}$	$0.000036_{(1)}$

Table 4
Computational time of recombination kernel.

Population	(1, 1)	(⌊|P|/2⌋, 1)	(1, min{1024, n})	(⌊|P|/2⌋, min{1024, n})
(30, 50)	$0.000502_{(4)}$	$0.000044_{(2)}$	$0.000053_{(3)}$	$0.000013_{(1)}$
(100, 500)	$0.015078_{(4)}$	$0.000323_{(3)}$	$0.000157_{(2)}$	$0.000029_{(1)}$
(1000, 1500)	$0.454974_{(4)}$	$0.004979_{(3)}$	$0.002161_{(2)}$	$0.000513_{(1)}$
(2000, 5000)	$3.013751_{(4)}$	$0.036462_{(3)}$	$0.012214_{(2)}$	$0.002581_{(1)}$

Table 5
Computational time of replacement kernel.

Population	(1, 1)	(1, min{1024, 2·|P|})
(30, 50)	$0.000594_{(2)}$	$0.000055_{(1)}$
(100, 500)	$0.006305_{(2)}$	$0.000163_{(1)}$
(1000, 1500)	$0.614006_{(2)}$	$0.001333_{(1)}$
(2000, 5000)	$2.476954_{(2)}$	$0.002686_{(1)}$

Table 6
Computational time of evaluation kernel over GF(x).

Matrix size	(1, 1)	(|P|, 1)	(1, min{1024, n})	(|P|, min{1024, n})
(30, 50)	$0.099388_{(4)}$	$0.004668_{(2)}$	$0.011550_{(3)}$	$0.000528_{(1)}$
(100, 500)	$31.407293_{(4)}$	$0.590733_{(2)}$	$1.282825_{(3)}$	$0.148725_{(1)}$

Table 7
Computational time of evaluation kernel over GF(2)

Matrix size	(1, 1)	(|P|, 1)	(1, min{1024, n})	(|P|, min{1024, n})
(30, 50)	$0.033808_{(4)}$	$0.001330_{(2)}$	$0.003239_{(3)}$	$0.000147_{(1)}$
(100, 500)	$2.457932_{(4)}$	$0.030540_{(2)}$	$0.099620_{(3)}$	$0.008287_{(1)}$

If we focus on results of the recombination threshold aggregation kernel in Table 3, we find that there are no significant differences between the sequential algorithm and the parallel kernel when the size of the population increases. In this case, it can be expected this behavior since the value to be aggregated is relatively small ($|P|$ = 2000), and the population size should be larger to effectively obtain benefits of parallelization in aggregation operation.

Considering the recombination kernel in Table 4, we can see that the proposed division $|P|/2$ blocks with n threads provides the best results in all cases. Each block is in charge of combining two parents to generate two children, and each thread is in charge of performing the crossover of a fraction of the parent chromosomes. The gain in execution time increases with the population and permutation sizes, ranging from 38,61 times faster for the smaller case to 1167,67 times faster for the largest setting. As recombination is a component that is executed in every iteration of the CHC algorithm, this improvement has a big impact in the whole execution time.

If we focus on the replacement kernel in Table 5, we can also verify a substantial gain in the parallel kernel compared to the sequential algorithm. For a small size problem, $|P|$ = 30, the gain in execution time is 10,8 times faster. For the largest population size, we obtain a gain in time complexity of 922,17 times faster for the parallel kernel. As this component is also executed every iteration of the algorithm, this is a great improvement for the overall algorithm's execution time.

Finally, we analyze the general version of the evaluation kernel for $GF(8)$ and its specific implementation for the binary case over $GF(2)$. We may observe in Table 6 that the evaluation is the most time consuming component of the algorithm, specially in the case of $GF(8)$. If the problem is relatively small, with matrix sizes of $G_{30,50}$, the execution time is affordable for a population size of $|P|$=30 in the sequential algorithm. However, if the problem size increases to matrices of size $G_{100,500}$, then the evaluation of $|P|$=100 individuals is very time consuming (in the order of seconds). The parallel version of the algorithm assumes that each block evaluates a single individual, and each thread in the block is in charge of performing all operations of a single matrix row. In the small case of $(30, 50)$, the parallel version provides a gain in execution time of 188,23 times faster. In the medium-size case of $(100, 500)$, the gain increases to 211,18 times faster, and makes experimentation affordable for larger codes.

Considering the special case of $GF(2)$, the matrices were implemented using bit-level representation, so that each row divides its size by the hardware word size 32. Also, finite field operations are implemented with AND and XOR bitwise logic operations. In this case, Table 7 shows a significant improvement in performance even in the sequential algorithm, which reduces the execution time of matrices of size $G_{100,500}$ from 31,4 s to 2,45 s. Also the best parallel execution reduces time from 0,15 s to 0,008 s. The gain in execution time from sequential to the parallel version holds in the range of 200–300 times faster than the sequential algorithm. As evaluation is a key component in the algorithm, then reducing the evaluation time from 31,4 s to the order of milliseconds is a significant improvement in our problem, which enables the possibility to study codes of larger size than using the sequential version of the algorithm.

To end up the discussion of the experimentation, we may conclude that parallelization of all components help to reduce the execution time of the complete algorithm. Moreover, a parallelization where each block performs independent operations and each thread within a block cooperates to accomplish the block objective is the best parallelization strategy for the problem addressed. Also, a simplification of the calculation of RREF in bit-wise operations can help to reduce the execution time of the evaluation kernel. This result is of special significance, as most of the studied codes in the literature are built over $GF(2)$.

4.2. Application to real scenarios

In this section, we compare the parallel version of CHC with the sequential algorithm. To do so, we propose a set of linear codes previously studied in Cuéllar et al. (2021). The sequential algorithm is also executed in GPU in a single core, to be able to compare the gain in time execution for the parallel proposal. The goal of this experimentation is to study the benefits of our approach as a method to find an upper

Table 8
Results over $GF(8)$ codes.

Code	Time (sequential)	Time (parallel)	Best solution	Time gain
$(30,16,10)_8$	0.901	0.004	$10_{856.0}$	217.33
$(30,18,9)_8$	0.985	0.004	$9_{856.0}$	238.95
$(45,22,15)_8$	1.679	0.005	$15_{871.38}$	331.92
$(45,24,14)_8$	1.825	0.005	$14_{871.38}$	363.36
$(45,26,12)_8$	1.932	0.005	$12_{871.38}$	370.13
$(45,28,11)_8$	2.054	0.005	$11_{871.38}$	385.40
$(60,28,21)_8$	2.953	0.007	$21_{846.28}$	404.57
$(60,30,20)_8$	3.211	0.008	$20_{846.28}$	420.50
$(60,32,19)_8$	3.471	0.008	$19_{846.28}$	433.21
$(60,34,17)_8$	3.699	0.009	$17_{846.28}$	413.31
$(75,30,28)_8$	4.372	0.010	$28_{861.24}$	424.29
$(75,35,24)_8$	7.551	0.015	$24_{1600.62}$	500.44
$(75,40,20)_8$	6.249	0.014	$20_{861.24}$	462.34
$(90,19,49)_8$	5.648	0.021	$49_{5474.38}$	267.52
$(90,50,21-22)_8$	233.877	0.558	$22_{100200.67}$	419.32
$(90,60,16)_8$	206.452	0.424	$16_{38588.56}$	487.31
$(90,70,11)_8$	13.427	0.028	$11_{865.93}$	472.08
$(130,85,23)_8$	682.217	1.520	$24_{100008.0}$	448.89
$(130,95,18)_8$	571.297	1.290	$18_{32317.12}$	442.90

Table 9
Results over $GF(2)$ codes.

Code	Time (integer)	Time (binary)	Best solution	Time gain
BCH $(511,103,123)_2$	6.837	0.777	128_{100000}	8.80
BCH $(511,121,117)_2$	9.055	0.848	136_{100000}	10.68
BCH $(511,166,95)_2$	14.441	1.205	119_{100000}	11.98
BCH $(511,76,171)_2$	0.226	0.025	$171_{1072.0}$	8.99
BCH $(511,58,183)_2$	0.139	0.018	$183_{703.31}$	7.55
EQR $(272,136,40)_2$	0.336	0.040	$40_{3227.1}$	8.36
EQR $(338,169,40)_2$	6.890	0.722	$40_{54474.58}$	9.54
EQR $(368,184,48)_2$	9.710	1.004	$48_{92005.0}$	9.67
EQR $(432,216,48)_2$	15.814	1.494	68_{100000}	10.59
EQR $(440,220,48)_2$	16.960	1.530	64_{100000}	11.08

bound of the distance of linear codes (or to confirm the designed distance in the best case), with special focus in time complexity. For this reason, a common setting is used among all problems studied. This setting was tested over all problems and it provided suitable results in average, in a preliminary experimentation. We used a population of $|P|=500$ individuals whose length n depends on each problem. The value τ to calculate the decrement value of recombination threshold is set to 0.1. There are two stopping criteria: Either to reach a maximum of 100.000 solutions evaluated in the algorithm, or to find the optimal code distance given by the known lower bound for each code. Finally, we performed 30 runs of the algorithms for each dataset to make statistical tests.

Two types of codes were used for the experimentation: 19 codes of increasing size using $GF(8)$ as underlying finite field, and 10 larger codes over $GF(2)$ to study the benefits of bit-wise parallel version of RREF. Table 8 shows the results of CHC execution over a single core and fully parallelized version. Column 1 describes the code as a tuple $(n,k,d)_q$, where n is the code length, k is the code dimension, q is the number of elements in the finite field, and d the known code distance. If a range d_1–d_2 is provided instead of d, then the true distance of the code is not known, but a lower/upper bound interval. We remark that most of the known code distances were obtained by the sequential version of the proposed method in Cuéllar et al. (2021). Columns 2–3 in Table 8 show the average time (in seconds) the algorithm spent to perform single-core execution and parallel execution, respectively. Column 4 describes the best solution found, and it is shown in subscripts the average number of evaluations required until the optimal/best solution was found. Finally Column 5 shows the gain in execution time from the sequential algorithm to the parallel proposal.

According to Table 8, we observe a substantial gain in execution time of the proposal compared to the sequential algorithm, even for small problems. Also, as it was expected, the optimal solution was found in most of the problems. The gain in execution time from the sequential to parallel version of the algorithm ranges from 200 to 500 times faster. This is a substantial increase which enables the possibility to study larger codes with the proposed technique. A non-parametric Wilcoxon test with 95% confidence level was applied to verify that there are significant differences between execution time of sequential and parallel algorithm designs.

On the other hand, Table 9 shows the results obtained to solve binary codes. We used five BCH codes widely used in the literature and five Extended QR codes (EQR). In this experiment, our goal is to test experimentally the gain of the binary bit-wise implementation of parallel RREF evaluation against the baseline parallel proposal implemented for any finite field with integer representation. The structure of Table 9 is the same as in Table 8. According to the results obtained, we may see that using a compact binary representation of the dataset, and XOR and AND bit-wise operations, can provide a substantial improvement in execution time. The gain ranges from 8.8 to 11.98 times faster from baseline parallel algorithm to the binary implementation, which suggests that the binary representation can obtain a gain in the order of thousands of times faster than the sequential baseline algorithm. In this case, the non-parametric Wilcoxon test with 95% confidence level was also applied to verify that there are significant differences between time complexity of binary and baseline methods.

To end up with the discussion of results, we also compare the proposal with the existing method from Canteaut and Chabaud (1998) in Table 10 regarding some BCH codes of length 511 for which a word of minimum weight is still unknown. For the comparison, we use the fast binary-encoding CHC algorithm run with a population of 10000 individuals and a stopping criterion of 100000000 solutions evaluated. Ten different experiments (algorithm executions) were performed in each case. Column 1 in Table 10 shows the code (word length, dimension and designed distance); Column 2 prints the average computing time for each execution; Column 3 describes the minimum weight word obtained by our approach; and Column 4 plots the reference distance found in Canteaut and Chabaud (1998, TABLE VII). We remark with (*) those codes for which our proposal was able to achieve a new result. In Canteaut and Chabaud (1998, Proposition 8) it is argued that, for a given BCH code with designed distance δ, if a word with even weight $w>\delta$ is found, then there exists a word of weight $w-1$. For this reason, the experiments in Canteaut and Chabaud (1998) were stopped when reaching a word of weight $\delta+1$. The number of codewords with weight $\delta+1$ is $(n-\delta)/(\delta+1)$ times greater that the number of codewords with weight δ. Also, it is conjectured that the true minimum distance of a BCH code does not exceed $\delta+4$. These authors obtained a codeword of weight 30 for the BCH code $(511,385,29)$ in less than a minute, and, in theory, it should takes around 16 min for finding a codeword with weight 29. Our method reached the distance in less than four minutes on average. They also found a word of weight 38 for the code $(511,358,37)$ in three days (theoretically, around 37 days for finding a word of minimum weight). Our method needed, on average, around 40 min for finding a word of weight 37. For the codes $(511,340,41)$ and $(511,193,87)$ we have not reached the minimum distance before the stopping criterion held, but we have found codewords with weight 42 and 88 in around 7 and 4 h, respectively. In Canteaut and Chabaud (1998), the authors assert that they needed between one and three weeks for these tasks. On the other hand, we have also found words of weight $\delta+4$ and $\delta+5$ for the BCH codes $(511,238,75)$ and $(511,148,107)$, respectively, and these results support the aforementioned conjecture about the distance of BCH codes. In the case of the code $(511,229,77)$, we found a new upper bound with a word of weight 83, although it is unsure if the true distance is under this value.

We remark some lessons learned after this piece of research. First, parallel design of algorithms must take into account the hardware architecture where they will be implemented, to fully optimize time gain. In our case, since the parallel proposal was targeted at GPU

Table 10
Comparison with some codes in Canteaut and Chabaud (1998, TABLE VII).

Code	Time (s.)	Best solution	Reference solution
BCH (511, 148, 107)	20311.23	112 (*)	Unknown
BCH (511, 193, 87)	15647.64	88	88
BCH (511, 229, 77)	2698.99	83 (*)	Unknown
BCH (511, 238, 75)	24604.12	79 (*)	Unknown
BCH (511, 340, 41)	24963.31	42	42
BCH (511, 358, 37)	2268.28	37 (*)	38
BCH (511, 385, 29)	221.10	29 (*)	30

implementation, the main principle we followed to build the design was to split the problem into separate non overlapped subproblems that will be solved among blocks, and to use threads within a block in cooperation to solve a specific subproblem. Also, global memory accesses can be a bottleneck in GPUs, so that the use of local and/or shared memory should be prioritized by threads whenever it is possible, to avoid concurrent use (mainly write operations) to the same memory region. Regarding the problem of finding the minimum distance of linear codes, our approach shows high potential with regards other previous works published in the literature, and more specifically the sequential algorithm of Cuéllar et al. (2021). On the other hand, we remark that the results provided by the algorithm must be considered as upper bounds of the true code distance but, when it is equal to the designed code distance, this is confirmed. The accuracy of the proposal to find the true minimum distance depends on the specific case being studied, but theoretical studies (Weishui & Chen, 1996; Raphaël, 1998) over the complexity class of approximated algorithms, and more specifically genetic algorithms, ensure that the global minimum can be achieved with non-zero probability as the population size and number of algorithm iterations increases.

5. Conclusions

In this work, we propose a GPU-based parallel model for the computation of Reduced Row Echelon Form and its integration within a complete parallel design of an adaptation of the CHC evolutionary algorithm. We analyze the time complexity of the resulting parallel components of the algorithm both in theory and practice, and conclude that the gain in execution time in practice may vary from nearly 100 to nearly 1000 times faster than a sequential algorithm, depending on the component of the algorithm. Moreover, we tested the approach in different real datasets considering linear codes over $GF(8)$ and $GF(2)$. In the latter case, a bit-wise operations implementation of the parallel RREF algorithm may provide a gain in execution time of thousands of times faster with respect to a sequential algorithm running integer representation of the code matrix. The results obtained in the experiments also suggest that the classic McEliece's cryptosystem (McEliece, 1978) could be vulnerable to an attack following this methodology. As a future work we may also consider the adjustment of our methods to other settings. For instance, convolutional codes comprise an important class of codes whose error-correcting capacity is measured with respect to the free distance. In Johannesson and Zigangirov (1999) it is showed an algorithm for computing the free distance based on the iterated calculation of two sequences related to the Hamming distance of certain block codes. Therefore, a suitable insertion of our techniques in this scheme could speed up the calculation of the free distance. Another important class of codes is the one formed by low density parity check (LDPC) codes. These are linear block codes with a parity check matrix containing a relatively few non-zero elements. Therefore, by adapting our algorithms to the calculation of the minimum distance through a parity check matrix, we could take advantages of this property and get good performance with respect to the run-time.

CRediT authorship contribution statement

Manuel P. Cuéllar: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **F.J. Lobillo:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Gabriel Navarro:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work has been supported by projects PID2019-110525GB-I00/AEI/10.13039/501100011033 and FEDER/Junta de Andalucía-Consejería de Economía *y* Conocimiento/A-FQM-470-UGR18

References

Askali, M., Azouaoui, A., Nouh, S., & Belkasmi, M. (2013). On the computing of the minimum distance of linear block codes by heuristic methods. *International Journal of Communications, Network and System Sciences*, *05*, http://dx.doi.org/10.4236/ijcns.2012.511081.

Barbieri, A., Cagnoni, S., & Colavolpe, G. (2005). Genetic design of linear block error-correcting codes. In B. Apolloni, M. Marinaro, & R. Tagliaferri (Eds.), *Biological and artificial intelligence environments* (pp. 107–116). Dordrecht: Springer Netherlands.

Berisha, A., Bytyçi, E., & Tershnjaku, A. (2017). Parallel genetic algorithms for university scheduling problem. *International Journal of Electrical and Computer Engineering*, *7*, 1096–1102. http://dx.doi.org/10.11591/ijece.v7i2.pp1096-1102.

Betten, A., Braun, M., Fripertinger, H., Kerber, A., Kohnert, A., & Wassermann, A. (2006). *Error-correcting linear codes volume 18 of algorithms and computation in mathematics*. Springer.

Bilbao, M., & Leguizamón, G. (2019). Multicore parallelization of chc for optimal aerogenerator placement in wind farms. In: *XXV Congreso Argentino de Ciencias de la Computación - CACIC 2019* (pp. 85–94).

Bland, J. (2007). Local search optimisation applied to the minimum distance problem. *Advanced Engineering Informatics*, *21*, 391–397. http://dx.doi.org/10.1016/j.aei.2007.01.002.

Canteaut, A., & Chabaud, F. (1998). A new algorithm for finding minimum-weight words in a linear code: application to mceliece's cryptosystem and to narrow-sense bch codes of length 511. *IEEE Transactions on Information Theory*, *44*, 367–378. http://dx.doi.org/10.1109/18.651067.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *10*, CALCULATEURS PARALLELES.

Casanova, H., Iacono, J., Karsin, B., Sitchinava, N., & Weichert, V. (2017). An efficient multiway mergesort for gpu architectures. *Preprint*.

Chen, S., Davis, S., Jiang, H., & Novobilski, A. (2011). Cuda-based genetic algorithm on traveling salesman problem. In R. Lee (Ed.), *Computer and information science 2011* (pp. 241–252). Berlin, Heidelberg: Springer Berlin Heidelberg.

Cuéllar, M., Gómez-Torrecillas, J., Lobillo, F., & Navarro, G. (2021). Genetic algorithms with permutation-based representation for computing the distance of linear codes. *Swarm and Evolutionary Computation*, *60*, Article 100797. http://dx.doi.org/10.1016/j.swevo.2020.100797.

Davidson, A., Tarjan, D., Garland, M., & Owens, J. D. (2012). Efficient parallel merge sort for fixed and variable length keys. In *2012 innovative parallel computing (InPar)* (pp. 1–9). http://dx.doi.org/10.1109/InPar.2012.6339592.

Dontas, K., & De Jong, K. (1990). Discovery of maximal distance codes using genetic algorithms. In *Proceedings of the 2nd international IEEE conference on tools for artificial intelligence* (pp. 805–811). http://dx.doi.org/10.1109/TAI.1990.130442.

Dumas, J.-G., Gautier, T., Pernet, C., & Sultan, Z. (2014). Parallel computation of echelon forms. In F. Silva, I. Dutra, & V. Santos Costa (Eds.), *Euro-Par 2014 parallel processing* (pp. 499–510). Cham: Springer International Publishing.

Eshelman, L. J (1991). The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundations of Genetic Algorithms*, *1*, 265–283. http://dx.doi.org/10.1016/B978-0-08-050684-5.50020-3.

Filipiak, P., & Lipiński, P. (2012). Parallel chc algorithm for solving dynamic traveling salesman problem using many-core gpu. In A. Ramsay, & G. Agre (Eds.), *Artificial intelligence: methodology, systems, and applications* (pp. 305–314). Berlin, Heidelberg: Springer Berlin Heidelberg.

García-Calvo, R., Guisado, J., del Rio, F. D., Córdoba, A., & Jiménez-Morales, F. (2018). Graphics processing unit–enhanced genetic algorithms for solving the temporal dynamics of gene regulatory networks. *Evolutionary Bioinformatics*, *14*, Article 1176934318767889. http://dx.doi.org/10.1177/1176934318767889.PMID:29662297.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, *29*, 147–160. http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x.

Huffmann, W. C., & Pless, V. (2003). *Fundamentals of error-correcting codes*. Cambridge University Press.

Johannesson, R., & Zigangirov, K. S. (1999). *Fundamentals of convolutional coding*. Wiley-IEEE Press.

Leon, J. S. (1988). A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, *34*, 1354–1359. http://dx.doi.org/10.1109/18.21270.

Liesen, J., & Mehrmann, V. (2015). Linear algebra. In *Springer undergraduate mathematical series*. Springer International Publishing, http://dx.doi.org/10.1007/978-3-319-24346-7.

Linton, S., Nebe, G., Niemeyer, A., Parker, R., & Thackray, J. (2018). A parallel algorithm for gaussian elimination over finite fields. arXiv, abs/1806.04211.

Lisonek, P., & Trummer, L. (2016). Algorithms for the minimum weight of linear codes. *Advances in Mathematics of Communications*, *10*, 195–207.

Luo, J., Baz, D. E., & Hu, J. (2018). Acceleration of a cuda-based hybrid genetic algorithm and its application to a flexible flow shop scheduling problem. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (pp. 117–122). http://dx.doi.org/10.1109/SNPD.2018.8441112.

Luo, J., Fujimura, S., El Baz, D., & Plazolles, B. (2019). Gpu based parallel genetic algorithm for solving an energy efficient dynamic flexible flow shop scheduling problem. *Journal of Parallel and Distributed Computing*, *133*, 244–257. http://dx.doi.org/10.1016/j.jpdc.2018.07.022.

McEliece, R. J. (1978). A public-key cryptosystem based on algebraic coding theory. *44*, *DSN Progress Report*, (pp. 114–116).

Mühlenbein, M., & Born, J. (1991). The parallel genetic algorithm as function optimizer. *Parallel Computing*, *17*, 619–632. http://dx.doi.org/10.1016/S0167-8191(05)80052-3.

Muxiang, Z., & Fulong, M. A. (1994). Simulated annealing approach to the minimum distance of error-correcting codes. *International Journal of Electronics*, *76*, 377–384. http://dx.doi.org/10.1080/00207219408925934.

Nesmachnow, S., Cancela, H., & Alba, E. (2010). Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, *15*, 685–701. http://dx.doi.org/10.1007/s00500-010-0594-y.

Nouh, S., & Belkasmi, M. (2013). Genetic algorithms for finding the weight enumerator of binary linear block codes. *International Journal of Applied Research on Information Technology and Computing*, *2*, http://dx.doi.org/10.5958/j.0975-8070.2.3.022.

NVIDIA, Vingelmann, P., & Fitzek, F. H. (2020). Cuda release: 10.2.89. URL https://developer.nvidia.com/cuda-toolkit.

Overbeck, R., & Sendrier, N. (2009). Code-based cryptography. In D. J. Bernstein, J. Buchmann, & E. Dahmen (Eds.), *Post-Quantum Cryptography* (pp. 95–145). Berlin, Heidelberg: Springer Berlin Heidelberg, http://dx.doi.org/10.1007/978-3-540-88702-7_4.

Pospichal, P., Jaros, J., & Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. In C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcazar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preuß, J. Togelius, & G. N. Yannakakis (Eds.), *Applications of evolutionary computation* (pp. 442–451). Berlin, Heidelberg: Springer Berlin Heidelberg.

Raphaël, Cerf (1998). Asymptotic convergence of genetic algorithms. *Advances in Applied Probability*, *30*(2), 521–550.

Sinha, R., Singh, S., Singh, S., & Banga, V. K. (2016). Accelerating genetic algorithm using general purpose gpu and cuda. *International Journal of Computer Graphics*, *7*, 17–30. http://dx.doi.org/10.14257/ijcg.2016.7.1.02.

Skorpil, V., Oujezsky, V., Cika, P., & Tuleja, M. (2019). Parallel processing of genetic algorithms in python language. *vol. 372*, In *2019 photonics electromagnetics research symposium - spring (PIERS-Spring)* (pp. 3727–3731). http://dx.doi.org/10.1109/PIERS-Spring46901.2019.9017332.

Stephen, B., Kureshi, I., Brennan, J., & Theodoropoulos, G. (2017). Exploring the evolution of big data technologies. In I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, & B. Maxim (Eds.), *Software architecture for big data and the cloud* (pp. 253–283). Boston: Morgan Kaufmann, http://dx.doi.org/10.1016/B978-0-12-805467-3.00014-4.

Umbarkar, D. A., & Joshi, M. (2013). Review of parallel genetic algorithm based on computing paradigm and diversity in search space. *ICTACT Journal on Soft Computing*, *3*, 615–622. http://dx.doi.org/10.21917/ijsc.2013.0089.

Vardy, A. (1997). The intractability of computing the minimum distance of a code. *IEEE Transactions on Information Theory*, *43*, 1757–1766. http://dx.doi.org/10.1109/18.641542.

Weishui, Wan, & Chen, Xiong (1996). Convergence theorem of genetic algorithm. *vol. 3*, In *1996 IEEE international conference on systems, man and cybernetics*: *Cat. No.96CH35929*, (pp. 1676–1681). Information Intelligence and Systems, http://dx.doi.org/10.1109/ICSMC.1996.565352.

Xie, W. W., & Ning, T. (2013). Genetic algorithm for panel cutting stock on cuda platform. *vol. 256*, In *Advances in manufacturing science and engineering* (pp. 2569–2575). Trans Tech Publications Ltd, http://dx.doi.org/10.4028/www.scientific.net/AMR.712-715.2569.