The Journal of Supercomputing (2023) 79:10335–10352
https://doi.org/10.1007/s11227-023-05058-w

Kemeny ranking aggregation meets the GPU

Noelia Rico[1] · **Pedro Alonso**[2] · **Irene Díaz**[1]

Accepted: 12 January 2023 / Published online: 7 February 2023

Abstract

Ranking aggregation, studied in the field of social choice theory, focuses on the combination of information with the aim of determining a winning ranking among some alternatives when the preferences of the voters are expressed by ordering the possible alternatives from most to least preferred. One of the most famous ranking aggregation methods can be traced back to 1959, when Kemeny introduces a measure of distance between a ranking and the opinion of the voters gathered in a profile of rankings. Using this, he proposed to elect as winning ranking of the election the one that minimizes the distance to the profile. This is factorial on the number of alternatives, posing a handicap in the runtime of the algorithms developed to find the winning ranking, which prevents its use in real problems where the number of alternatives is large. In this work we introduce the first algorithm for the Kemeny problem designed to be executed in a Graphical Processing Unit. The threads identifiers are codified to be associated with rankings by means of the factorial number system, a radix numeral system that is then used to uniquely pair a ranking with the thread using Lehmer's code. Results guarantee constant execution time up to 14 alternatives.

Keywords Kemeny problem · Factorial system · Lehmer code · Graphical Processing Unit (GPU) · Parallelization

✉ Noelia Rico
noeliarico@uniovi.es

Pedro Alonso
palonso@uniovi.es

Irene Díaz
sirene@uniovi.es

[1] Department of Computer Science, University of Oviedo, Oviedo, Spain

[2] Department of Mathematics, University of Oviedo, Oviedo, Spain

1 Introduction

Due to the broad usage of decision-making tools nowadays in many contexts of our daily lives, how to develop programs that are able to make this decision is a hot topic in many research areas. In this kind of contexts, situations where dealing with the preferences of multiple voters over a set of alternatives is needed are frequently found. When voters express their preferences by ranking the potential options from most to least favored, the process of making a decision about the winning ranking is known as *ranking aggregation*, and it is a subfield of social choice theory [1, 2]. Recently, computational social choice has gain attention [3], motivated among other reasons by studying efficient algorithms that implement some ranking aggregation methods that, although properly mathematically defined, are not computationally efficient so they cannot be applied in real contexts.

A prominent example of one of these methods is the one proposed by Kemeny [4]. He began by proposing a distance, between a ranking and the opinion of the voters, to measure the disagreement between the ranking and the preferences. Using this, he proposed that the ranking with the shortest distance to the profile (among all the possible rankings defined over the set of alternatives) must be chosen as the election's winner. This method soon gained attention, as it was proved [5] to be the only ranking aggregation method that is at the same time neutral, consistent and moreover ranks in the best position the alternative that wins all others pairwisely (known as Condorcet winner) if it exists, some very desirable properties for any ranking aggregation method. Due to these properties the method would be suitable to be used in many situations, but this is prevented by its computational complexity. Finding the solution as proposed by Kemeny constitutes a NP-hard [6] problem, which makes it not appropriate for its use in most of the real situations, especially in presence of time constraints.

Therefore, the execution time when the number of alternatives is large represents a huge problem. For example, if the sequential computation of the distance of each ranking to the profile takes 1 second for 6 alternatives, then the time for 10 alternatives would be 40 min. Again, as it is factorial, this is even more noticeable when the number is large: for example, from 13 to 14 alternatives the execution time would change from approximately 9 days to 2 months. Due to these differences, the study of algorithms that aim to solve this problem is very important to make the method useful in real contexts. For this reason, even improvements that allow to increase one single alternative are a great contribution to the research field.

In the last years, Azzini and Munda [7] introduced an exact algorithm that greatly improved the runtime in relation to others published before [8, 9]. We have improved these results in [10, 11], ensuring the feasibility of the execution up to 13 alternatives. However, the runtime of the *branch and bound algorithms (B &B)* proposed in these works depends on the characteristics of the input profiles and cannot be exactly known prior to the execution. Notice that, there is an obvious dependence between the runtime and the number of alternatives of the profile and, moreover, the nature of branching algorithms makes that, even for problems with the same size, the real execution of the algorithm is sometimes possible and sometimes not, as it depends on the solutions pruned from the search space, which cannot be determined prior to the execution itself. This implies that the behavior of these algorithms varies widely even for different profiles that share the same number of alternatives and voters [12–14].

All the previously mentioned algorithms are designed and developed to be executed in a Central Processing Unit (CPU). However, the computational field is conditioned by the development in the hardware tendencies. Currently, the use of Graphical Processing Units (GPUs) for general parallel computing tasks, known as GPGPU after General Programming GPU, has provided a platform for investigating methods that leverage concurrent threads to tackle complex problems in computational terms. Nevertheless, how to translate classical CPU algorithms to GPU algorithms is not straightforward for many reasons. For example, CPU algorithms, especially those based on techniques such as B &B, are complex in the management of memory, and GPUs have more memory restrictions than CPUs despite being faster on parallel tasks. Moreover, some more problems appear when the development paradigm changes. Indeed, for combinatorial problems such as finding the Kemeny ranking, one of the main issues regards the storage space and how to handle the flow of data between CPU and GPU. Also the paradigm for the development of the code changes from CPU to GPU.

In this work, we propose an algorithm to be executed in a NVIDIA GPU using CUDA and Numba for Python. On the one hand, CUDA [15] is a compute platform that allows developers to run code in a massively parallel fashion on NVIDA GPUs. On the other hand, Numba [16] is a just-in-time Python function compiler that provides a tool for compiling Python code to CUDA for NVIDIA GPUs. The aim of this work is to introduce the first algorithm for the Kemeny problem developed to be executed in a GPU. This poses some main questions such as how to distribute the rankings in parallel and how to handle the management of the memory both in CPU and GPU. We propose to use thread identifiers, associating them with rankings by means of the factorial number system, which is a radix numeral system. Then, we use this factorial codification to uniquely pair a ranking with a thread using Lehmer's code. The considerations about the memory space that must be taken into account for the feasibility of this algorithm are also addressed.

The remaining of the document is structured as follows. In Sect. 2 the ranking aggregation problem is formally introduced, as well as the Kemeny method. Section 3 gives the explanation related to the codification systems used for paralleling and gives the algorithm proposed. The results obtained are discussed in Sect. 4. Final conclusions are drawn in the last section.

2 Ranking aggregation

Consider a set of n alternatives $\mathcal{A} = \{a_1, \ldots, a_n\}$ and a set of m voters expressing their preferences over $\mathcal{A}$. In this work, the scenario where preferences over $\mathcal{A}$ are expressed in the form of rankings is considered. We use the term ranking to refer to any strict order relation defined over $\mathcal{A}$, such that, for every pair of alternatives $a_i, a_j \in \mathcal{A}$, with $a_i \neq a_j$, a strict ($a_i \succ a_j$ or $a_j \succ a_i$) relation is defined.

The multiset π^n_m of rankings given by the m different voters over the set of n alternatives is called **profile of rankings**. As some voters may agree on their ranking, if all the rankings were listed this would lead to a representation of the profile containing repeated rankings. Assuming that anonymity of the voters is ensured, we use the compact representation of the profile of rankings that only contains $m' \leq m$ different rankings, where each ranking $r_i \in \pi^n_m$ is weighted by the number w_i of voters that expressed the ranking r_i. Thus, $m = \sum_{i=1}^{m'} w_i$.

The alternatives in $\mathcal{A}$ may be also compared in a pairwise fashion by using a matrix $\mathbf{O}$ of dimension $n \times n$, known as the **outranking matrix** [17]. Each element o_{ij} of $\mathbf{O}$, with $1 \leq i,j \leq n$, represents the number of voters that prefer alternative a_i over alternative a_j. The value of the element o_{ij} is obtained from π^n_m by adding 1 point every time that $a_i \succ a_j$ appears in a ranking of the profile. Therefore, for each pair of alternatives a_i and a_j with $i \neq j$, it holds that $o_{ij} + o_{ji} = m$. By definition, all the elements of the diagonal are set to 0. This representation contains the pairwise information provided by the profile of rankings and, although the preference orders are lost, it is enough to compute the Kemeny distance from a ranking to a profile as will be later explained in Eq. 1.

Ranking aggregation functions are used to summarize the preferences in the profile of rankings π^n_m in such way that the ranking chosen as winner represents a *consensus ranking*. How this consensus is reached is not trivial and has been and still is deeply studied in the field of social choice theory.

Condorcet [18] stated that an alternative a_i should be ranked at a better position than another alternative a_j in the winning ranking if a_i is preferred by the majority of the voters over a_j, which in terms of the outranking matrix means that $o_{ij} > o_{ji}$. Using this idea, Condorcet proposed a function to aggregate the preferences that results in a ranking where each alternative wins by majority every other alternative ranked in a worse position. The ranking that fulfills these characteristics is called *Condorcet ranking*. Accordingly, the alternative in the first position of the Condorcet ranking is known as Condorcet winner, i.e., the alternative that wins by majority in a pairwise comparison against any other alternative. Unfortunately, the relations given by the voters are not necessarily transitive as they may lead to situations in which $a_i \succ a_j$, $a_j \succ a_k$ and $a_k \succ a_i$, even if the preferences were expressed in the form of strict orders. This is famously known as *voting paradox*. When the voting paradox occurs, there is not Condorcet ranking. Nevertheless, sometimes it is possible to find a Condorcet winner even in absence of a Condorcet ranking. A ranking aggregation method is called a Condorcet method if it finds as winning alternative the Condorcet winner if it exists.

A prominent family of ranking aggregation functions is one based on the use of a distance function δ on the set of rankings. The distance of a ranking s to a profile of rankings $\delta(s, \pi^n_m)$ is computed by adding the individual distances from s to all rankings in π^n_m. From all the possible $n!$ complete rankings that can be obtained by permuting the set of n alternatives $\mathcal{A}$, the one (or ones) that minimizes the value of δ is selected as the winning ranking.

Table 1 Profile of rankings π^4_{10} given by ten voters on the set of four alternatives $\mathcal{A} = \{A, B, C, D\}$ (left) and corresponding outranking matrix (right)

Number of voters	Ranking		A	B	C	D
3	$A \succ B \succ C \succ D$	A	0	5	7	3
2	$D \succ B \succ A \succ C$	B	5	0	10	6
2	$D \succ A \succ B \succ C$	C	3	0	0	6
3	$B \succ C \succ D \succ A$	D	7	4	4	0

Table 2 Distance δ for all the possible rankings according to the Kemeny method for the profile of rankings in Table 1

Ranking	δ	Ranking	δ	Ranking	δ	Ranking	δ
$A \succ B \succ C \succ D$	23	$B \succ A \succ C \succ D$	23	$C \succ A \succ B \succ D$	37	$D \succ A \succ B \succ C$	23
$A \succ B \succ D \succ C$	25	$B \succ A \succ D \succ C$	25	$C \succ A \succ D \succ B$	39	$D \succ A \succ C \succ B$	33
$A \succ C \succ B \succ D$	33	$B \succ C \succ A \succ D$	27	$C \succ B \succ A \succ D$	37	$D \succ B \succ A \succ C$	23
$A \succ C \succ D \succ B$	35	$B \succ C \succ D \succ A$	23	$C \succ B \succ D \succ A$	33	$D \succ B \succ C \succ A$	27
$A \succ D \succ B \succ C$	27	$B \succ D \succ A \succ C$	**21**	$C \succ D \succ A \succ B$	35	$D \succ C \succ A \succ B$	35
$A \succ D \succ C \succ B$	37	$B \succ D \succ C \succ A$	25	$C \succ D \succ B \succ A$	35	$D \succ C \succ B \succ A$	37

The most representative example of this family of distance-based methods is the one proposed by Kemeny [19], who established a method based on the Condocet principle that uses distances in order to reach a solution if cycles are present in the profile.

2.1 Kemeny ranking rule

According to Kemeny, the distance between two rankings is the number of discrepancies in the relative order of every pair of alternatives. Formally, 1 point is added to the distance every time that two alternatives appear in the ranking in reverse order. Thus, the distance of a ranking s to the profile of rankings π^n_m is defined as the sum of the Kemeny distances from s to all the rankings $r_i \in \pi^n_m$.

The distance δ from a ranking s to a profile represented by the outranking matrix $\mathbf{O}$ can be computed using this matrix such that, having

$$x_{ij} = \begin{cases} o_{ji} & \text{if } a_i \succ a_j \\ o_{ij} & \text{otherwise} \end{cases}$$

the Kemeny distance is the sum of the values

$$\delta(s, \pi^n_m) = \sum_{i=1}^{n} \sum_{j=i+1}^{n-1} x_{ij} \,. \tag{1}$$

For the profile of rankings in Table 1, the distances from all possible rankings on the set of alternatives $\mathcal{A} = \{A, B, C, D\}$ are shown in Table 2. The distance is minimized by the ranking $B \succ D \succ A \succ C$. Therefore, this is the Kemeny ranking, i.e., the solution to the Kemeny problem for the profile of rankings in Table 1.

Unfortunately, as it was previously mentioned, the problem of finding a Kemeny ranking has been proved to be NP-hard [6, 20]. This means that it does not exist an algorithm to compute the Kemeny ranking in polynomial time for any number of alternatives.

3 Developing a Kemeny algorithm for the GPU

Considering the definition given by Kemeny to find the consensus of the winning ranking, any algorithm to find the Kemeny ranking can be roughly summarized into the following steps:

Step 1 For each possible strict ranking obtained as a permutation of n alternatives: compute the distance from the ranking to the profile of rankings given by the voters.
Step 2 Find the minimum distance obtained.
Step 3 Keep as winners only those strict rankings whose distance is equal to the minimum distance found after evaluating all the possible solutions.

Currently, in the literature there are several algorithms that solve the Kemeny ranking. These are implemented for CPUs [7, 9, 11]. The ones showing the best results are those that are designed as branch and bound (B &B) algorithms. B &B algorithms list the possible solutions to a problem in a tree structure known as search space. Then, they use rules to prune off regions of the search space that cannot lead to an optimal solution, thus avoiding the exploration of all the possible solutions, which notably can reduce the runtime. However, this reduction depends on the rules defined to prune the tree and also on the characteristics of the input profile. For this reason, the runtime that will be saved in relation to the full exploration of the possible solutions cannot be known in advance, and sometimes it could be so small that could prevent the execution of the algorithm in real time.

In this work it is presented a different approach for solving the Kemeny problem by developing a GPU algorithm instead of CPU algorithms. This algorithm looks for the winner ranking by executing multiple threads in parallel, where each thread takes care of computing the distance from different rankings to the profile in parallel. This saves time that requires to execute sequential tasks in the CPU but it comes with the cost that each thread is independent of the computations done by other thread.

The steps presented for the basic algorithm must be divided in smaller pieces such that some of them can be parallelized and done independently. Step 1 can be parallelized as for each ranking the same independent task must be done. However, when focusing on Step 2, it is necessary to know the computations done by the other threads in order to determine the winner.

As the cost of copying the rankings from the CPU to the GPU would be very expensive, the first step in order to define a GPU algorithm is to determine how to make each thread capable of knowing for which ranking must compute the distance without explicitly knowing the ranking first. The second step is to determine how to combine the information obtained by each thread and reduce it.

Moreover, there are some key implementation aspects that are usually not considered in the definition of the algorithm but when dealing with such large amount of information are extremely important to take into account in order to develop a suitable algorithm for the GPU. In contrast to happens in the CPU, where the use of arrays that require dynamic memory is very common, the memory required by the GPU to perform all the operations must be known in advance. This implies that copies of memory must be simplified as minimum. Moreover, the memory space of the GPU is limited, therefore, it is recommendable to reuse the arrays and mind the data types.

3.1 Mapping rankings to threads

As previously stated, the first step of the algorithm is to determine how each thread knows the ranking for which it must compute the distance. This can be divided in three different tasks:

1. Associate each thread with its factorial number (see Sect. 3.1.2).
2. Use the factorial number of the thread to get a ranking (see Sect. 3.1.3).
3. Compute the distance of the ranking to the profile using Eq. 1.

3.1.1 Ranking codification

A strict ranking can be understood as a numeric vector of non-repeated numbers, where each element of the vector is a natural number in the interval $[0, n-1]$ that represents the position of the alternative in the ranking, being 0 the best possible position and consequently $n-1$ the worst possible one. The codification proposed represents with the number 0 the alternative in the best position due to computational convenience. The alternative of $\mathcal{A}$ in the ith position of vector representing the ranking r, with $i \leq 0 < n$, is denoted by $r(i)$ and can be referred by its index j in the set $\mathcal{A}$ with $0 \leq j < n$. For example, given a set of alternatives $\mathcal{A} = \{A, B, C, D\}$ the ranking $C \succ A \succ D \succ B$ can be represented by the vector below show in Fig. 1.

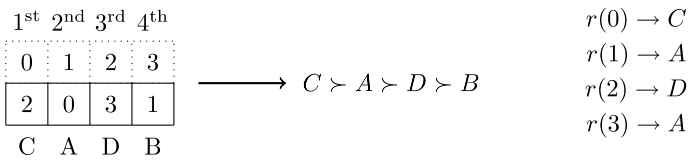

Fig. 1 Representation of a ranking as a vector. Given a set of n elements, the ith element of the vector stores the numeric representation of the alternative that is in that position of the ranking. Thus, the lower the index of an alternative in the vector, the better an alternative is ranked

The GPU automatically associates a unique integer identifier (henceforth id) to each thread in the grid of all the possible threads. The algorithm proposed in this work takes advantage of this to codify the id as factorial number that can be later associated with a ranking.

3.1.2 Factorial system

The factorial number system [21, 22], also known as *factoradic*, is a mixed radix numeral system. It is specially useful in combinatorics due to its property for numbering permutations [23].

In order to convert a natural number (in decimal base representation) to a factorial representation, it is necessary to obtain the sequence of digits in a factorial base. For a number lower than $n!$ are required n digits. Thus, using a vector of n elements, the most-left element corresponds with the times that $(n-1)!$ must be multiplied, the next element with $(n-2)!$ and so on. For example, the vector (2, 0, 1, 0) represents the number $2 \times 3! + 0 \times 2! + 1 \times 1! + 0 \times 0! = 13$. This can be expressed also as $2{:}0{:}1{:}0_!$ to denote the factorial base.

Algorithm 1 shows how to formally compute the factorial number from an integer number.

Algorithm 1 Factoradic codification

```
1: $x = n$
2: radix = 1
3: while $x \neq 0$ do
4:     $f_{n-\text{radix}} = \text{remainder}\left(\frac{x}{\text{radix}}\right)$
5:     $x = \text{quotient}\left(\frac{x}{\text{radix}}\right)$
6:     radix = radix + 1
7: end while
```

An example of how to apply Algorithm 1 is shown in Table 3. This illustrates how to compute the factorial number 789. First of all, the vector $\mathbf{f}$ to represent this number must have at least $n = 7$ positions, as $7! > 789 > 6!$. The vector $\mathbf{f}$ is filled from right to left with the reminder obtained from the consecutive operations. In the first iteration $i = 0$, the number is divided by the radix = 1, obtaining as results the same number and the reminder 0, that is used for filling the last element of $\mathbf{f}$. Then, the radix is incremented and the quotient of the last operation, which is again the initial number, is divided by 2, and the obtained remainder 1 added to $\mathbf{f}_{n-\text{radix}}$. In the result of the previous operation, 394, is divided by radix = 3, and so on, until the result obtained is 0. The vector $\mathbf{f} = (0, 0, 2, 3, 1, 1, 0)$ obtained as the factorial number corresponding to 789 can be separated as $0 \times 6! + 0 \times 5! + 2 \times 4! + 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 789$. Notice how, as the most-right element corresponds always with the reminder of dividing the number by 1, this value is always 0, so it could be omitted in the representation to save memory space.

Table 3 Example of the iterations needed for getting the factorial number of 789

i	quotient	radix	result	reminder	factoradic
0	789	1	789	0	0
1	789	2	394	1	1 0
2	394	3	131	1	1 1 0
3	131	4	32	3	3 1 1 0
4	32	5	6	2	2 3 1 1 0
5	6	6	1	0	0 2 3 1 1 0
6	0	7	0	0	0 0 2 3 1 1 0

3.1.3 Lehmer's code for GPU

The factorial number can be converted into a ranking using the numbers as inversion table. The aim is to be able to do the translation of a factorial number as shown in Fig. 2. In this work, we choose to translate the factorial number as ranking applying Lehmer's code.

Lehmer [24] proposed an encoding for each possible permutation of a sequence of n numbers. As there are $n!$ factorial permutations, each one of them can be used to represent one of the $n!$ rankings over a set of alternatives. To obtain Lehmer's code from a factorial representation, the position of the alternatives in the set of n values is considered (starting in 0). The alternative in the position of the first number of the factorial representation is taken out of the set. After taking the first alternative from the set of n elements, the indexes are updated and the next alternative is obtained from a fixed set of $n-1$ elements, and so forth decreasing the number of possibilities until the last number for which only a single fixed value is allowed.

The steps to get a ranking r from a factorial number $\mathbf{f}$ using Lehmer's code are the following:

- Consider that the elements in $\mathcal{A}$ are numerated from 0 to $n-1$ and the factorial number is defined by the vector $\mathbf{f}$, also indexed from 0 to $n-1$.

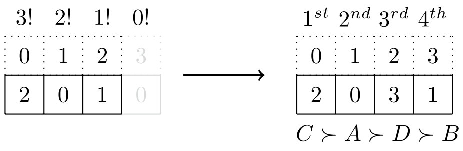

Fig. 2 Example of the ranking obtained over the set of alternatives $\mathcal{A} = \{A, B, C, D\}$ from a factorial number

- For $i \in [0, n)$, do:

  1. Take from $\mathcal{A}$ the alternative x that is in position f_i in the set.
  2. Update $r(i) = x$.
  3. Remove x from $\mathcal{A}$.
  4. Renew the indexes of the alternatives in $\mathcal{A}$.

Table 4 exemplifies how to get a ranking from the factorial number $1{:}1{:}0{:}0_!$. The first column is the iteration number i; the second column represents the factorial number $\mathbf{f}$, highlighted in yellow the i-th alternative being evaluated at each iteration i and crossing the alternatives that have been already evaluated; the third column shows the set of the alternatives $\mathcal{A}$ (which are represented by its position in the set, i.e., 0=A,1=B,2=C,3=D) and above them its corresponding index in $\mathcal{A}$ for each iteration is shown; the ranking r resulting in each iteration appears in the fourth column. The status of the vectors with the factorial number and the set of alternatives after the iteration are shown in the 5th and 6th columns.

A natural implementation in a CPU for this procedure would be to use dynamic memory for having a vector of variable length representing the set of alternatives. Unfortunately, developing code for GPU kernels is much more restricted. For example, the allocation of the memory must be static, meaning that the portion of memory that must be allocated is required to be known at compile time. Thus, we propose Algorithm 2 for obtaining Lehmer's code from the factorial number $\mathbf{f}$, which uses a Boolean vector to keep track of the alternatives that have been already added to the ranking.

The Boolean vector $\mathbf{a}$ is used to mark the alternative, which also avoids to reindex them manually, as a counter is used to keep track of the positions. In line 1, the iteration over all the elements of the set begins. Then it is required to iterate until the value f_i is reached, which determines the element of the set of alternatives in that position that has

Table 4 Example on how to obtain a ranking from a factorial number using Lehmer's code

				After the iteration	
i	Number $\vec{f}$	Alternatives	Ranking	Number $\vec{f}$	Alternatives
0	$\begin{array}{cccc}1&1&0&0\end{array}$	$\begin{array}{cccc}0&1&2&3\\0&1&2&3\end{array}$	$\begin{array}{cccc}1&&&\end{array}$	$\begin{array}{cccc}\cancel{1}&1&0&0\end{array}$	$\begin{array}{cccc}0&&1&2\\0&\cancel{1}&2&3\end{array}$
1	$\begin{array}{cccc}\cancel{1}&1&0&0\end{array}$	$\begin{array}{cccc}0&&1&2\\0&\cancel{1}&2&3\end{array}$	$\begin{array}{cccc}1&2&&\end{array}$	$\begin{array}{cccc}\cancel{1}&\cancel{1}&0&0\end{array}$	$\begin{array}{cccc}0&&&1\\0&\cancel{1}&\cancel{2}&3\end{array}$
2	$\begin{array}{cccc}\cancel{1}&\cancel{1}&0&0\end{array}$	$\begin{array}{cccc}0&&&1\\0&\cancel{1}&\cancel{2}&3\end{array}$	$\begin{array}{cccc}1&2&0&\end{array}$	$\begin{array}{cccc}\cancel{1}&\cancel{1}&\cancel{0}&0\end{array}$	$\begin{array}{cccc}&&&0\\\cancel{0}&\cancel{1}&\cancel{2}&3\end{array}$
3	$\begin{array}{cccc}\cancel{1}&\cancel{1}&\cancel{0}&0\end{array}$	$\begin{array}{cccc}&&&0\\\cancel{0}&\cancel{1}&\cancel{2}&3\end{array}$	$\begin{array}{cccc}1&2&0&3\end{array}$	$\begin{array}{cccc}\cancel{1}&\cancel{1}&\cancel{0}&\cancel{0}\end{array}$	$\begin{array}{cccc}&&&\\\cancel{0}&\cancel{1}&\cancel{2}&\cancel{3}\end{array}$

not been explored yet. This is the element that must be considered to be added to the ranking. To make this possible, line 2 defines a counter, to keep track of the alternatives that have been explored. Also, the alternative that will be added is stored in j. Lines 5 to 10 iterate over the vectors looking for the alternative to add following this idea. When this alternative is found, it is marked as already added in the Boolean vector **a** (line 11) and added to the ranking in line 12.

Algorithm 2 Lehmer's code algorithm

```
Require: a vector with the factorial number $\vec{f}$
1: Define the vector $\vec{a} = (0, \ldots, 0)$        ▷ To mark the alternatives added
2: for $i \in \{0, \ldots, n-1\}$ do                  ▷ For each alternative of the set
3:     $c = 0$                                        ▷ Counter to check when $f_i + 1$ is reached
4:     $j = 0$                                        ▷ The index of the alternative being evaluated
5:     while $c < f_i + 1$ do                         ▷ Iterate until the threshold is reached
6:         if $a_j$ is 0 then                         ▷ If it is not already in the ranking
7:             $c = c + 1$                            ▷ Increment the counter of alternative considered
8:         end if
9:         $j = j + 1$                                ▷ Prepare for the next iteration
10:    end while
11:    $a_{j-1} = 1$                                  ▷ Mark the alternative as used
12:    $f_i = j - 1$                                  ▷ Reuse $f$ to store the ranking
13: end for
```

Notice how in the algorithm proposed the vector **f** is overwritten over the factorial representation, as shown in line 12 of the algorithm. This means that column 2 and 4 in Table 4 are in fact the same vector in memory. This drastically saves memory space, helping the feasibility of the implementation as, otherwise, the size of the grid would be duplicated, which makes impossible the execution of the algorithm due to hardware restrictions. To illustrate this, let us refer by 'grid' to a matrix stored in memory where each row corresponds to the codification of one factorial number (i.e., each vector **f**) or one ranking (as the codification shown in Fig. 2). Therefore, if this grid is defined for $n = 4$, this must have $4! = 24$ rows and 4 columns, i.e., a total of 96 elements to be able to store the codifications for all the possible rankings with 4 alternatives. If each element of the grid is codified as an unsigned integer of 8 bits (i.e., using 1 byte), the size of the grid in memory is equal to 96 bytes (i.e., 96B). Following the same idea, it would require 600B for 5 elements, 4.22KB for 6 elements and so on. To calculate the number of bytes required in memory to store the grid when the value of n is increased one unit, the following expression can be used:

$$n * \mu(n-1) + n! \,. \tag{2}$$

Considering that $\mu(n-1)$ is the memory required by the grid of $n-1$ alternatives, as for the exponential nature of the grid the size of $n-1$ must be multiplied n times, and then one more column must be added to all rows, i.e., $n!$ elements of 1 byte (so the $\times 1$ factor can be skipped in the equation).

Following the example where each element is of 1 byte, for 12 alternatives the memory for the grid is up to 5.35 GB. The fact that the algorithm avoids to create one grid for the factorial codification and another for the ranking obtained using Lehmer's code, and instead uses codifications that allows to create a single grid which elements can be overwritten to obtain both codification as explained in Algorithm 2, saves in this case more than 5GB of RAM memory.

An example of Algorithm 2 is shown in Table 5, where the iterations to get a ranking from the number $1{:}1{:}0{:}0_!$, that represents the number 8 are detailed until the ranking (1, 2, 0, 3) is obtained. This ranking represents $B \succ C \succ A \succ D$ over the set of alternatives $\mathcal{A} = \{A, B, C, D\}$.

Table 5 Detailed trace of an example using Algorithm 2

i	f_i	c	j	$c < f_i + 1$	a_j	Action	Explored status a	f	Line
0	1	0	0	True	0	Update $c \to c = 1$	$\boxed{0}\boxed{0}\boxed{0}\boxed{0}$	$\boxed{1}\boxed{1}\boxed{0}\boxed{0}$	
						Update $j \to j = 1$			9
		1	1	True	0	Update $j \to j = 2$			9
		1	2	False	0	Do not enter the loop			5
						Update $a_0 \to 1$	$\boxed{0}\boxed{1}\boxed{0}\boxed{0}$		11
						Update $f_0 \to 1$		$\boxed{1}\boxed{1}\boxed{0}\boxed{0}$	12
1	1	0	0	True	0	Update $c \to c = 1$			7
						Update $j \to j = 1$			9
		1	1	True	1	Update $j \to j = 2$			9
		1	2	True	0	Update $c \to c = 2$			5
						Update $j \to j = 3$			9
		2	3	False	0	Do not enter the loop			5
						Update $a_1 \to 1$	$\boxed{1}\boxed{1}\boxed{0}\boxed{0}$		11
						Update $f_1 \to 2$		$\boxed{1}\boxed{2}\boxed{0}\boxed{0}$	12
2	0	0	0	True	1	Update $j \to j = 1$			9
		0	1	True	1	Update $j \to j = 2$			9
		0	2	True	0	Update $c \to c = 1$			7
						Update $j \to j = 3$			9
		1	1	False	0	Do not enter the loop			5
						Update $a_2 \to 0$	$\boxed{1}\boxed{1}\boxed{0}\boxed{0}$		11
						Update $f_2 \to 0$		$\boxed{1}\boxed{2}\boxed{0}\boxed{0}$	12
3	0	0	0	True	1	Update $j \to j = 1$			9
		0	1	True	1	Update $j \to j = 2$			9
		0	2	True	1	Update $j \to j = 3$			9
		0	3	True	0	Update $c \to c = 1$			7
						Update $j \to j = 3$			9
		0	4	False	0	Do not enter the loop			5
						Update $a_3 \to 0$	$\boxed{0}\boxed{0}\boxed{0}\boxed{0}$		11
						Update $f_3 \to 3$		$\boxed{1}\boxed{2}\boxed{0}\boxed{3}$	12

3.2 Final algorithm

Physical restrictions of the hardware are usually omitted when algorithms are presented. However, when developing an algorithm that deals with such amount of data, hardware comes as the principal restriction in terms of memory capability. For this reason, it is necessary to design the algorithm taking into account the memory limitations of the hardware. One of these limitations has been already mentioned in the previous section, as it is necessary to reuse the grid employed for the representation of the factorial number to later store the rankings, from which the distance must be computed. A theoretical idea of this implementation is that showed in Fig. 3 for the set of alternatives $\mathcal{A} = \{A, B, C\}$, where each thread computes a distance for a single ranking.

However, the implementation of the idea shown in this figure is not feasible in reality. A main hardware restriction concerns the size of the grid to store the factorial representation and the rankings. For this reason, a *stride* is introduced in the algorithm to have control over the number of threads, recycling each thread to compute those rankings which factorial number can be codified using a multiple of the id of the thread. This means that each thread do not take care only of the ranking associated which its id, but also all those that are associated with the integer number (id+(stride$\times i$)) $< n!$ with i taken the value of the natural numbers until the condition is not fulfilled anymore. Thus, the size of the grid to store the codifications can be reduced to stride$\times n$, and the vector corresponding to each thread is overwritten iteratively. The value of the stride can be set prior to the execution and allows the code to be executed effectively by GPUs with different architectures and capabilities.

Note also that it is not feasible (neither useful nor necessary) to keep all the distances. Therefore, it is used a general register to store the δ^*, the best distance, which initially is $+\infty$. The threads access this variable to check whether the distance they are computing improves the one stored there. In case this is true, the variable is updated and the id of the ranking that gives the new distance is considered as solution.

The final algorithm is presented in Algorithm 3. The process is launched in the CPU, which transfer the preferences to the GPU to compute the solution in parallel following the algorithms presented, and then the solution reached is given back to the CPU so it can be returned to the user. The CPU is the intermediary between the

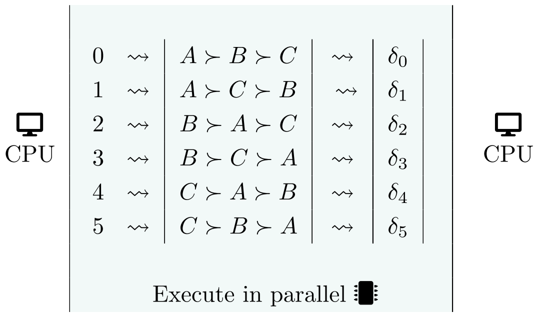
CPU
0 ⇝ $A \succ B \succ C$ ⇝ δ_0
1 ⇝ $A \succ C \succ B$ ⇝ δ_1
2 ⇝ $B \succ A \succ C$ ⇝ δ_2
3 ⇝ $B \succ C \succ A$ ⇝ δ_3
4 ⇝ $C \succ A \succ B$ ⇝ δ_4
5 ⇝ $C \succ B \succ A$ ⇝ δ_5
Execute in parallel
CPU

Fig. 3 Theoretical diagram of the GPU codification

user of the algorithm and the GPU. This algorithm has been developed in Python 3.8 and the Numba [16] package, which provides an interface for developing code in CUDA for NVIDIA GPUs [15]. The code that implements the algorithm is publicly available in http://github.com/noeliarico/kemenyGPU.

Algorithm 3 Kemeny on GPU

```
1: Start the process in CPU.
2: Define the size of the stride.
3: Initialize the grid on the GPU of dimension stride $\times n$.
4: Copy the outranking matrix $\mathbf{O}$ in the GPU.
5: Launch the process in parallel
6: for each thread $i$ do                                ▷ In parallel
7:     $j = i$
8:     while $j < n!$ do
9:         Compute $\vec{f_j}$, the factorial representation of the integer $j$ (see
           Algorithm 1).
10:        Compute the ranking $r_j$ from the factorial representation $\vec{f_j}$ using
           Lehmer's code (see Algorithm 2).
11:        Compute the distance from $r_j$ to the profile of rankings using $\mathbf{O}$
           (see Eq. 1).
12:        if $\delta_{r_j} < \delta^*$ then
13:            Update $\delta^* = \delta_{r_j}$.
14:            Empty the list of solutions.
15:            Add $j$ to the list of solution.
16:        end if
17:        if $\delta_{r_j} == \delta^*$ then
18:            Add $j$ to the list of solution.
19:        end if
20:        $j = j +$ stride
21:    end while
22: end for
23: Copy the winning distance to the CPU.
24: Copy the solution to the GPU.
25: Decode the solution using the factoradic system and Lehmer's code.
```

4 Experiments and results

The algorithm proposed has been tested for a total of 19600 profiles of rankings. These profiles have been also used in the work [11], so using them provides a fair comparison between the results obtained for the CPU algorithm in relation to the ones obtained with the GPU algorithm proposed in this work. The list of profiles of rankings have been synthetically generated as follows:

- For each number of alternatives $n \in [8, 14]$:
  - For each number of alternatives in $m \in \{10, 50, 100, 250, 500, 1000, 2000\}$ and the same numbers plus 1.
    1. Randomly select the number $d \leq m$ of different rankings in the profile.
    2. Obtain d random different permutations of the n alternatives.
    3. A random vector of d elements whose sum is equal to m is generated where each element represents the number of voters associated to each ranking generated in the second step.
    4. If the profile does not have a Condorcet winner, create the outranking matrix and add this to the list.

When a Condorcet ranking exists, the solution of the Kemeny method for the ranking aggregation problem is this Condorcet ranking, which is straightforward to compute and thus the application of the algorithm is not required. This makes the profiles that do not have a Condorcet ranking more 'difficult' to solve.

The GPU algorithm has been tested using a NVIDIA GPU model GeForce RTX 3090. The runtime for each profile of rankings has been measured three times, keeping the median value as runtime of the profile. The results obtained for all the profiles with $n = 13$ keeps constant no matter the characteristics of the profile of rankings with a runtime around ~15 s, never exceeding the threshold of ~17 s. Moreover, for $n = 14$ alternatives, the runtime in seconds of any profile are in the interval [~42, ~47], with a mean result for all the profiles of rankings with any number of voters ~45 s. The variation in the runtime is justified as it can occurs that other processes may interfere in the communication between the CPU and the GPU, slightly slowing the copy of data.

In comparison with the results obtained for the CPU in [11], in this case the execution times in CPU are achieved due to the bounds of the algorithm, which makes the algorithm very variable and unpredictable as the execution time cannot be known in advance. The best runtimes in CPU for $n = 13$ gives similar results than the GPU but is very affected by the number of voters and whether this is odd or even and also slowing the algorithm when it increases. The improvement in the GPU algorithm is shown for $n = 14$, as for the CPU algorithm it is not guaranteed that the solution can be reached. The results obtained for 14 alternatives in the CPU are only for those profiles with low number of voters, for which the algorithm is faster, and even some of those can unpredictably overpass the two minutes of execution time. Moreover, for the CPU algorithm is highlighted that, considering profiles of the same number of alternatives, the number of voters impact on the execution time, even if the dimension of the outranking matrix is the same for any number of n.

It is important to highlight that, apart from the reduction in the execution times provided by the GPU, the greatest advantage is that the runtimes obtained with the GPU are invariable for any profile with any number voters when fixing the number of alternatives, in contrast of what happens with the CPU version. This means that the GPU can guarantee the execution time for any profile if another profile of the same number of alternatives have been previously executed.

5 Conclusion

In this work, we present a solution to the aggregation problem based on a GPU codification. This is a novel work in the field of social choice theory, introducing modern programming techniques to the field. The results show an advantage in relation to the previous implementation of the Kemeny method in CPU, ensuring the execution times for profiles of rankings with 14 alternatives around ∼45 s. Moreover, the runtimes of the GPU are constant and not affected by the characteristics of the profile of rankings, as happened in the CPU algorithms, which performance relies in the characteristics of the input profile of rankings. Furthermore, this comparison is made between a CPU algorithm that already include many restriction to reduce the search space and the first approach done in GPU, which is a brute force approach with room for improvement. This shows a promising path in this research line, as the algorithm can be further studied to improve, for example designing how to include rules to reduce the search space that have been previously proved useful for CPU algorithms.

We consider that the main contribution of this work is the modeling of the codification, associating each thread with a ranking by means of the factorial representation and the Lehmer code to solve the Kemeny problem. Also, the codification of both steps sequentially into the same grid is important, as ensures the feasibility of the implementation. This is not a trivial translation from a brute force implementation in CPU, and the combination of the codifications proposed provides a setting for further research in parallel algorithms to solve the Kemeny problem. Moreover, the codification chosen for the ranking codification ease the study of the inclusion of bounds based on theoretical restrictions. Our motivation for choosing this encoding and no other is that Lehmer's code lists the rankings in lexicographical order, which makes possible to locate sections of consecutive rankings that have a common top alternatives and discard them from the search, in the fashion of B &B algorithms, which is not trivial as the reduction in the runtime is not so easily achieved [25].

Author Contributions N.R. designed and developed the algorithm, carried the experiments and wrote the manuscript. P.A. reviewed the manuscript. I.D supervised the work and reviewed the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research has been partially supported by grant TIN2017-87600-P from the Spanish Government. Noelia Rico is supported by the Severo Ochoa program of the Principality of Asturias with the grant PA-20-PF-BP19-167.

Data Availability Statement http://github.com/noeliarico/kemenyGPU.

Declarations

Conflict of interest The authors declare they have no conflict of interests to report regarding the present work.

Ethical approval Not applicable.

References

1. Young HP (1988) Condorcet's theory of voting. Am Political Sci Rev 82(4):1231–1244
2. Pérez-Fernández R, Rademaker M, Alonso P, Díaz I, Montes S, De Baets B (2016) Representations of votes facilitating monotonicity-based ranking rules: From votrix to votex. Int J Approx Reason 73:87–107
3. Brandt F, Conitzer V, Endriss U, Lang J, Procaccia AD (2016) Handbook of computational social choice. Cambridge University Press, Cambridge, UK
4. Kemeny JG (1959) Mathematics without numbers. Daedalus 88(4):577–591
5. Young HP, Levenglick A (1978) A consistent extension of Condorcet's election principle. SIAM J Appl Math 35(2):285–300. https://doi.org/10.1137/0135023
6. Bartholdi J, Tovey CA, Trick MA (1989) Voting schemes for which it can be difficult to tell who won the election. Soc Choice Welfare 6(2):157–165
7. Azzini I, Munda G (2020) A new approach for identifying the Kemeny median ranking. Eur J Oper Res 281:388–401
8. Muravyov SV (2013) Ordinal measurement, preference aggregation and interlaboratory comparisons. Measurement 46:2927–2935
9. Amodio S, D'Ambrosio A, Siciliano R (2016) Accurate algorithms for identifying the median ranking when dealing with weak and partial rankings under the Kemeny axiomatic approach. Eur J Oper Res 249(2):667–676
10. Rico N, Vela CR, Pérez-Fernández R, Díaz I (2021) Reducing the computational time for the Kemeny method by exploiting Condorcet properties. Mathematics. https://doi.org/10.3390/math9121380
11. Rico N, Vela CR, Díaz I (2022) Reducing the time required to find the Kemeny ranking by exploiting a necessary condition for being a winner. Eur J Oper Res. https://doi.org/10.1016/j.ejor.2022.07.031
12. Ali A, Meila M (2012) Experiments with Kemeny ranking: What works when? Math Soc Sci 64(1):28–40
13. Betzler N, Fellows MR, Guo J, Niedermeier R, Rosamond FA (2009) Fixed-parameter algorithms for Kemeny rankings. Theor Comput Sci 410(45):4554–4570
14. Rico N, Vela CR, Díaz I (2022) Runtime bounds prediction for the Kemeny problem. J Ambient Intell Humanized Comput. https://doi.org/10.1007/s12652-022-03881-2
15. Tuomanen B (2018) Hands-On GPU programming with Python and CUDA: explore high-performance parallel computing with CUDA. Packt Publishing Ltd, Birmingham, UK
16. Lam SK, Pitrou A, Seibert S (2015) Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, pp 1–6
17. Arrow K, Raynaud H (1986) Social choice and multicriterion decision-making, 1st edn. The MIT Press, Cambridge (MA), USA
18. Condorcet M (1785) Essai sur l'Application de l'Analyse à la Probabilité des Décisions Rendues à la Pluralité des Voix. De l'Imprimerie Royale, Paris
19. Kemeny JG (1959) Mathematics without numbers. Daedalus 88(4):577–591
20. Hemaspaandra E, Spakowski H, Vogel J (2005) The complexity of Kemeny elections. Theoret Comput Sci 349(3):382–391

21. Laisant C-A (1888) Sur la numération factorielle, application aux permutations. Bulletin de la Société Mathématique de France. 16:176–183. https://doi.org/10.24033/bsmf.378
22. Knuth D (1969) Seminumerical Algorithm. The Art of Computer Programming, vol. 2. Addison–Wesley, Reading (MA), USA
23. Marmion M-E, Regnier-Coudert O (2015) Fitness landscape of the factoradic representation on the permutation flowshop scheduling problem. In: Dhaenens C, Jourdan L, Marmion M-E (eds) Learning and intelligent optimization. Springer, Cham, pp 151–164
24. Lehmer DH (1960) Teaching combinatorial tricks to a computer. In: Proceedings of Symposia in Applied Mathematics Combinatorial Analysis, vol. 10, pp 179–193
25. Han TD, Abdelrahman TS (2011) Reducing branch divergence in gpu programs. In: Proceedings of the fourth workshop on general purpose processing on graphics processing units. GPGPU-4. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1964179.1964184