A SIMD algorithm for the detection of epistatic interactions of any order

Christian Ponte-Fernández *, Jorge González-Domínguez, María J. Martín

Universidade da Coruña, CITIC, Computer Architecture Group, 15071 A Coruña, Spain

ABSTRACT

Epistasis is a phenomenon in which a phenotype outcome is determined by the interaction of genetic variation at two or more loci and it cannot be attributed to the additive combination of effects corresponding to the individual loci. Although it has been more than 100 years since William Bateson introduced this concept, it still is a topic under active research. Locating epistatic interactions is a computationally expensive challenge that involves analyzing an exponentially growing number of combinations. Authors in this field have resorted to a multitude of hardware architectures in order to speed up the search, but little to no attention has been paid to the vector instructions that current CPUs include in their instruction sets. This work extends an existing third-order exhaustive algorithm to support the search of epistasis interactions of any order and discusses multiple SIMD implementations of the different functions that compose the search using Intel AVX Intrinsics. Results using the GCC and the Intel compiler show that the 512-bit explicit vector implementation proposed here performs the best out of all of the other implementations evaluated. The proposed 512-bit vectorization accelerates the original implementation of the algorithm by an average factor of 7 and 12, for GCC and the Intel Compiler, respectively, in the scenarios tested.

1. Introduction

Epistasis is the interaction of genetic variation at two or more loci during the expression of a phenotype that cannot be attributed to the additive combination of effects corresponding to the individual loci [1]. It is a phenomenon present in many plant [2,3], animal [4,5] and human [6,7] traits. Because of its importance, epistasis detection has been, and currently is, a topic under active research.

Besides its biological implications, epistasis also represents a computational challenge: locating a combination of features (or loci) that can correctly classify the samples (or individuals) in different groups attending to its phenotype, between all of the possible combinations of features. For this reason, a multitude of methods have been proposed in order to solve this problem, many of which refrain from exploring every combination and, instead, implement non-exhaustive alternatives ranging from greedy algorithms to machine learning techniques (see, for instance, [6,8–11]). In a previous study [12], we compared the performance of all epistasis detection methods published during the last decade that offer an implementation available to the scientific community, when locating epistatic interactions of different orders. The study concludes that, despite the rich variety of methods, only the exhaustive approaches (those which explore every combination of loci up to a certain combination size) can reliably identify interactions with no marginal effects.

Given the computational complexity of finding epistatic interactions, authors in this field have resorted to a multitude of different architectures to perform this task, which include CPUs [6,13,14], clusters of CPUs [15,16], GPUs [14,17,18], clusters of GPUs [16] and other accelerators [19,20]. Most modern CPU architectures, if not all, include Vector Processing Units (VPUs) in their processing cores. Exploiting all the resources available in a core is key in order to achieve the maximum performance. Although compilers incorporate automatic vectorization techniques to exploit the VPUs in programs that do not make explicit use of them, they show limitations on what can be automatically vectorized, and the performance obtained is not always the optimal, as it will be later seen. Vectorization has already been successfully employed in other bioinformatic applications such as local sequence alignment [21,22] or genome and metagenome distance estimation [23]. However, despite the potential that an efficient use of these units offers, none of the works previously mentioned, with the exception of [14], consider their use to further accelerate the epistasis search. In [14] Campos et al. include a couple of Intel Advanced Vector Extensions (AVX) Intrinsics to parallelize the bitwise *AND* and *AND-NOT* operations, used during the computation of the genotype frequencies corresponding to a particular loci combination.

* Corresponding author.
E-mail address: christian.ponte@udc.es (C. Ponte-Fernández).

In this work, we detailedly explore the intricacies of the SIMD parallelization of an exhaustive epistasis detection algorithm to find interactions of any order, looking to maximize the performance per core in modern CPUs. To do this, we manually implement an epistasis search explicitly using vector instructions, and compare it with the performance achieved by the GCC and Intel compilers automatically vectorizing the same operations. Starting from an existing method for third-order epistasis detection (MPI3SNP [16]), Section 2 describes how this method can be expanded to support loci combinations of any size. We selected MPI3SNP due to its good performance shown when locating epistasis [12] and the simplicity of the algorithm implemented. Section 3 explains how each of the operations can be vectorized through AVX Intrinsics using the *AVX2* and *AVX512BW* extensions. Section 4 presents the experimental evaluation of the vectorized algorithm, comparing the performance achieved by the explicit vector implementations using AVX Intrinsics with the performance achieved by the automatic vectorization that compilers offer and the original performance of the MPI3SNP implementation. Finally, Section 5 discusses the conclusions extracted from this study, reflects on its limitations and comments on some lines of future work.

2. Exhaustive epistasis detection

This work explores the extension of the exhaustive third-order epistasis detection algorithm used by MPI3SNP [16] to support the detection of epistasis interactions of any order, as well as the optimization of its implementation from the perspective of its single-thread performance.

Exhaustive epistasis detection methods, despite their differences in implementation, follow a common set of steps summarized in the flowchart shown in Fig. 1. The key elements of the program flow are:

1. The enumeration of all combinations of loci, without repetition, for a set of loci and up to a certain combination size.
2. The computation of the different allele combination frequencies for each combination of loci.
3. The quantification of the association between the division of the individuals in groups (such as cases and controls), and the differences in genotype frequencies between those groups, for each combination.

In this work, a depth-first algorithm is used for the enumeration of combinations of any given size, genotype and contingency tables are used for the computation of the different allele frequencies and the Mutual information (MI) method is employed for the quantification of the association between groups.

Genotype tables were first introduced by Wan et al. in BOOST [13], and since then have been adopted by a variety of epistasis detection works, including MPI3SNP [16] (some other examples are [14,18,24]). Genotype tables are a data structure used to represent the genotype information of a particular locus or combination of loci, following a binary format that allows for the combination with further loci using binary operators exclusively. This work uses Single Nucleotide Polymorphisms (SNPs) as the input genotype information. An SNP represents a specific locus in the genome where at least 1% of the population presents a genomic variation.

Genotype tables, once calculated, can be transformed into contingency tables that represent the allele combination frequencies of all individuals by simply counting bits on the genotype table rows. Wan et al. present the genotype and contingency tables as a single operation. However, when exploring combinations of
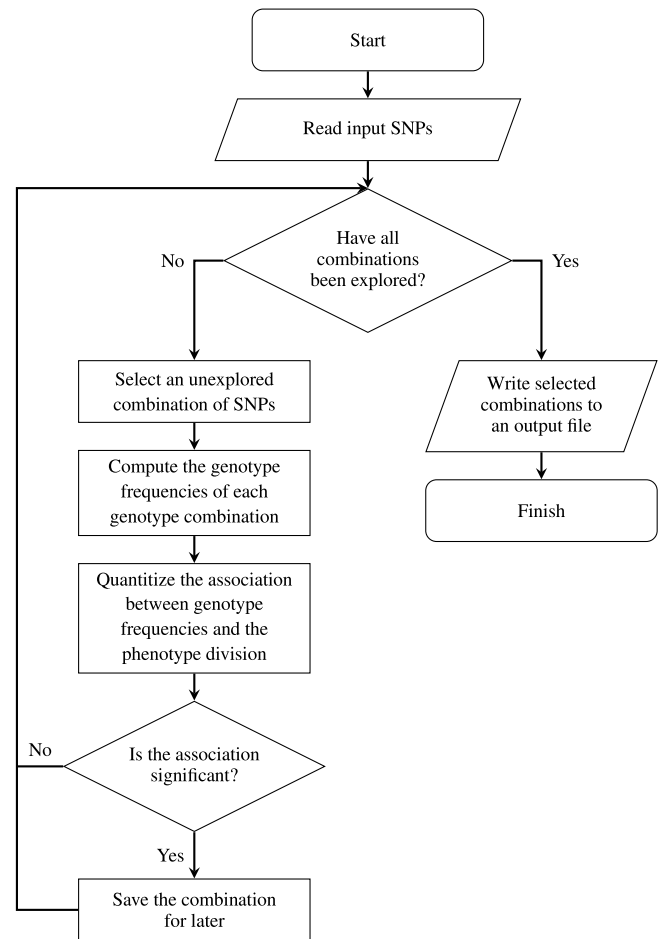

Fig. 1. Flowchart of a typical exhaustive epistasis detection method.

more than two loci, as is the case here, it is convenient to separate both operations since many combinations share a common number of loci with one another, and thus many intermediate genotype tables can be used to compute the contingency tables of different combinations. Sections 2.1 and 2.2 cover the calculation of genotype and contingency tables for combinations of variable size, respectively.

Once the contingency table of a particular loci combination is calculated, its association with the phenotype of interest can be measured using MI. It is a metric from Information Theory that measures the amount of information obtainable from one variable through the observation of another. Section 2.3 covers the calculation of this metric. Lastly, after all the operations are defined, Section 2.4 describes the algorithm that combines these operations to exhaustively explore every combination of loci for a particular combination size and locate the ones most associated with the phenotype.

2.1. Genotype table calculation

Genotype tables are data structures used to encode the genotype information following a binary format. Genotype tables group individuals by their phenotype class into two separate subtables, one for cases and another for controls. These subtables contain as many rows as genotypes each individual can have, and as many bits in each row as individuals represented in the table. When representing a single SNP of a biallelic population, as is the case with human populations, each individual can

Genotype table of the SNP i:

	Cases	Controls
i_1	01011010 00001111	01010010 10001111
i_2	00000101 00100000	10000100 01100000
i_3	10100000 11010000	00101001 00010000

Genotype table of the SNP j:

	Cases	Controls
j_1	00101011 10010101	00100001 00111001
j_2	10010000 00000000	11001010 10000100
j_3	01000100 01101010	00010100 01000010

Genotype table of SNP $i \times$ SNP j:

	Cases	Controls
$i_1 \cap j_1$	00001010 00000101	00000000 00001001
$i_1 \cap j_2$	00010000 00000000	01000010 10000100
$i_1 \cap j_3$	01000000 00001010	00010000 00000010
$i_2 \cap j_1$	00000001 00000000	10000000 00000000
	⋮	⋮
$i_3 \cap j_3$	00000000 01000000	00000000 00000000

Fig. 2. Example of two genotype tables of two different SNPs, i and j, for sixteen cases and controls, and the combined genotype table of the two SNPs.

have three different genotypes, encoded in three different rows: homozygous dominant, heterozygous and homozygous recessive. A 0 or a 1 in a particular position of a row indicates the absence or presence of the genotype corresponding to that row for the individual represented in that position, respectively. Using this representation, the information of a singular SNP for m individuals can be encoded in $3m$ bits. Fig. 2 includes two examples of genotype tables for two different SNPs i and j, both of which include 32 individuals and are encoded using 96 bits.

Genotype tables simplify the operation of combining different SNPs, as they can also be used to represent the combination of multiple genotypes. Considering that each row of the table represents the presence of a particular genotype in every individual as set bits, one can find which individuals have a particular genotype combination by calculating the intersection between the corresponding table rows. Therefore, obtaining the genotype distribution of a particular SNP combination involves combining the rows of the different tables and calculating their intersection via bitwise *AND* operations, for the case and control subtables separately. This procedure can be used to combine as many SNPs as necessary, resulting in a genotype table containing 3^s rows and $3^s m$ bits in total, with s being the number of SNPs in combination and m the number of individuals represented. Fig. 2 also includes an example of this, in which the two previous example SNPs i and j are combined into a genotype table containing a total of 9 rows.

Note that in an exhaustive exploration of combinations up to a certain size, smaller combinations are bound to appear in many larger ones. In this work, to maximize the reuse of tables and, thus, the performance, instead of combining numerous SNPs at once, genotype tables are constructed following an iterative process, adding one SNP to an existing table in each step. Additionally, and in contrast with the original method in BOOST [13], we are decoupling the representation of the genotype information in tables from contingency tables representing numeric genotype frequencies, as the latter loses the information of singular individuals and the frequencies are not needed in every step of the algorithm.

Listing A.1 shows a candidate C++ implementation of this operation in function `combine`, taking as arguments a genotype table representing the combination of any number of SNPs, a genotype table of a singular SNP and a genotype table where the results will be stored. Note that the template argument `uint64_t`, common to all genotype table classes, indicates the type used to store the binary information (Lines 18–20). Since the x86_64 instruction set operates with 64-bit integers, this type is ideal to hold the genotype information, each value representing the information of 64 individuals, and each computation operating with 64 individuals at once. The function calls the `combine_subtable` subroutine twice to combine each of the two subtables for cases and controls. This function consists of three nested *for* loops, the two outermost loops (Lines 6 and 7) combine the different rows of the input genotype tables. The innermost loop (Line 8) iterates over the different `uint64_t` values of the selected rows, reading one value from each input table, calculating its intersection and storing the result in the output table. The computational time complexity of this operation is $O(3^s m)$, where m is the number of individuals in the data and s is the size of the combination.

2.2. Contingency table calculation

Contingency tables represent the frequency distribution of two discrete variables from a number of observations. For this domain of application, the variables represented in the contingency tables are the phenotype and genotype variation. Constructing these tables from the genotype table representation is direct, as individuals are already segregated into different subtables according to the case and control groups, and different rows according to their genotype information. Obtaining the frequency distribution can be done by counting how many bits are set to one for each row. This operation is known as population count (or, in short, *popcount*), and most of the current processors provide hardware support for this operation.

Separating the genotype table and contingency table calculations into completely different operations may seem convenient at first glance, as this allows for genotype tables common to multiple SNP combinations to be reused. However, if we compute them entirely separate, the resulting values from the computation of the genotype table are stored in memory just to be brought back immediately after, in order to compute the corresponding contingency table. Because of this, it is more convenient to compute the rows of the last-level genotype table and, instead of storing them as usual, perform the *popcount* operation in a single step, saving us from several load and store instructions.

Listing A.2 shows a C++ implementation for this operation, very similar to the genotype table calculation (Listing A.1), showing that the function `combine_and_popcnt` consists of two calls to the subroutine `popcnt_subtable` to compute the contingency subtables of the two new genotype subtables, and this subroutine consists of the same three nested loops. However, instead of immediately storing the multiple `uint64_t` values resulting from the bitwise *AND* operations, the *popcount* operation is called, the results of the same row are summed up in a single `uint32_t` value and the sum is stored in the contingency table (Lines 10–12). In contrast with the genotype table class, the contingency table class uses the `uint32_t` type (passed as a template argument in Line 21) to represent the total count of individuals having a particular genotype because it can contain a large enough integer, and for the convenience of matching the size of a `float` value which will be useful later during the vectorization of the MI operation. The computational time complexity of this operation is also $O(3^s m)$, with m being the number of individuals in the data and s the size of the combination.

2.3. Mutual information calculation

MI quantifies the degree of association between the genotype distribution of cases and controls and the phenotype distribution obtained from the previously calculated contingency table. MI has shown a very good detection power in our previous comprehensive study [12], and the presence of low frequencies in the data, which become more prevalent as we move towards larger combination sizes, do not seem to be a problem.

MI is a metric from Information Theory that measures the amount of information obtainable from one variable through the observation of another one. Taking the genotype and phenotype variability as two random variables X and Y, MI can be obtained as:

$$MI(X; Y) = H(X) + H(Y) - H(X, Y) \qquad (1)$$

where $H(X)$ and $H(Y)$ are the marginal entropies of the two variables, and $H(X, Y)$ is the joint entropy. Marginal entropies of a single and two variables are defined as:

$$H(X) = - \sum_{x \in X} p(x) \log p(x) \qquad (2)$$

$$H(X, Y) = - \sum_{x, y} p(x, y) \log p(x, y) \qquad (3)$$

with $p(x)$ representing the probability of the random variable X taking the value x, $p(y)$ the probability of the random variable Y taking the value y, and $p(x, y)$ the joint probability of both events. These probabilities can be obtained directly from the contingency table as the division between the number of occurrences and the number of total observations.

Listing A.3 shows a base C++ implementation of the function to calculate the MI from an existing contingency table. It computes $H(X, Y)$ and $H(X)$ in a single *for* loop (Lines 11–18 and 19–22, respectively). The loop includes three *if* branches to avoid computing the logarithm of 0, which would lead to an undefined product of $0 \times -\infty$, resulting in a NaN value. $H(Y)$ and the inverse of the number of individuals (iinds) are provided as function arguments because they are independent of the genotype distribution of individuals, and thus can be calculated just once outside the function (Lines 3 and 4). The MI function operates with float types since single-precision floating point numbers offer enough numerical precision to represent the MI values. The time complexity of this operation is $O(3^s)$, where s is the size of the combination represented in the input contingency table.

2.4. Combinatory exploration algorithm

Algorithm 1 shows the pseudocode of a depth-first exploration algorithm which relies on the genotype table, contingency table and MI functions previously defined to combine the different SNPs of the input data and assess the degree of association between the SNP combinations and the phenotype of study. The key element of this algorithm is that it iterates over all the combinations in a depth-first manner with the help of a stack. This is fundamental to prevent multiple calculations of the same genotype table, since combinations may share a common set of SNPs with other combinations. When the combination space is explored depth-first, we exhaust all combinations starting with a particular prefix (and its corresponding genotype table) before moving onto the next one.

The arguments to this routine are the data set d containing the genotype tables of all individual SNPs in the data, the order o of the interactions to locate and a vector v where the results will be returned. In the first three lines, the function starts by allocating enough space for an array g of $o - 2$ genotype tables

---

Algorithm 1: Non-segmented exploration

Input:
d: Data set containing n genotype tables, each representing a single SNP for m individuals
o: Order of the interactions to identify
Output:
v: Vector of combinations of o SNPs, and their associated MI value

1 Allocate an array g of $o - 2$ genotype tables, for sizes between 1 and $o - 1$
2 Allocate a contingency table c of size o
3 Create an empty stack s
4 Calculate the marginal entropy $H(Y)$
5 $inv\_inds \leftarrow 1/m$
6 **for** $i \leftarrow 0$ **to** n **do**
7 $\quad$ $g[1] \leftarrow d[i]$
8 $\quad$ **for** $j \leftarrow i + 1$ **to** n **do**
9 $\quad\quad$ Push the pair $\{i, j\}$ into the stack s
   $\quad$ **end**
10 $\quad$ **while** s is not empty **do**
11 $\quad\quad$ Pop the combination $\{c_1, \ldots, c_k\}$ from the top of the stack s
12 $\quad\quad$ **if** $k < o$ **then**
13 $\quad\quad\quad$ $g[k] \leftarrow combine(g[k-1], d[c_k])$
14 $\quad\quad\quad$ **for** $j \leftarrow c_k + 1$ **to** n **do**
15 $\quad\quad\quad\quad$ Push the new combination $\{c_1, \ldots, c_k, j\}$ into the stack s
   $\quad\quad\quad$ **end**
   $\quad\quad$ **end**
16 $\quad\quad$ **else**
17 $\quad\quad\quad$ $c \leftarrow combine\_and\_popcnt(g[o-1], d[c_k])$
18 $\quad\quad\quad$ $mi\_val \leftarrow MI(c, H(Y), inv\_inds)$
19 $\quad\quad\quad$ Add $\{c_1, \ldots, c_k\}$ and mi_val to the vector of results v
   $\quad\quad$ **end**
   $\quad$ **end**
   end

---

of size 1 to $o - 1$, a contingency table c for combinations of the target size o and a stack s of combinations of SNP indexes. Before starting the exploration, the function computes the inverse of the number of individuals inv_inds, and the entropy of the phenotype variability $H(Y)$ (Lines 4 and 5), the two arguments of the MI function common to all combinations.

After that, the function starts to loop through all SNPs, exploring all of the combinations starting with that SNP before moving onto the next one. To do this, the genotype table of the SNP i is copied in $g[1]$, and all combinations of two SNPs starting with that one are pushed into the stack (Lines 7–9). Then, using a *while* loop, the combinations of the stack are processed until it is emptied. In each iteration, the top combination $\{c_1, \ldots, c_k\}$ of the stack is popped (Line 11). If k is smaller than the target interaction order o, its corresponding genotype table is computed from the genotype table of its prefix (stored in the array G) and the table of the last SNP c_k (Line 13). Then, all subsequent combinations starting with $\{c_1, \ldots, c_k\}$ are pushed into the stack (Lines 14–15). Otherwise, if k is equal to o, its contingency table and MI are computed, and the result is stored in the vector of results v (Lines 17–19).

For simplicity, in the pseudocode, we are appending all combinations with its MI to the vector of results, although in the actual implementation only the combinations with the highest MI value are retained in the vector. The computational time complexity of this algorithm is $O\left(\binom{n}{o} 3^o m\right)$, where n is the number of SNPs in the data, m is the number of individuals in the data, and o is the size of the combinations explored.

Table 1

Elapsed time, in seconds, during the computation of a single contingency table using different operation widths and *popcount* implementations. The table highlights with green background the best times using the *AVX512BW* extension, and with red text the best times using the *AVX2* extension.

AND width	POPCNT width	POPCNT algorithm	Individuals count					
			256	512	1024	2048	4096	8192
512	512	Harley seal	8.60×10^{-7}	8.60×10^{-7}	1.02×10^{-6}	1.31×10^{-6}	1.88×10^{-6}	1.60×10^{-6}
512	512	Lookup	2.47×10^{-7}	2.47×10^{-7}	3.39×10^{-7}	5.57×10^{-7}	9.63×10^{-7}	1.78×10^{-6}
512	256	cpu	4.04×10^{-7}	4.04×10^{-7}	6.91×10^{-7}	1.18×10^{-6}	2.20×10^{-6}	4.19×10^{-6}
512	256	Harley seal	7.59×10^{-7}	7.59×10^{-7}	1.05×10^{-6}	1.61×10^{-6}	1.72×10^{-6}	2.82×10^{-6}
512	256	Lookup	2.98×10^{-7}	2.98×10^{-7}	4.64×10^{-7}	7.66×10^{-7}	1.40×10^{-6}	2.67×10^{-6}
512	256	Lookup orig.	2.99×10^{-7}	2.99×10^{-7}	4.57×10^{-7}	8.01×10^{-7}	1.46×10^{-6}	2.81×10^{-6}
512	64	Popcnt movdq	3.02×10^{-7}	3.02×10^{-7}	5.19×10^{-7}	9.99×10^{-7}	1.86×10^{-6}	3.60×10^{-6}
512	64	Popcnt un. err.	4.36×10^{-7}	4.36×10^{-7}	7.09×10^{-7}	1.17×10^{-6}	2.06×10^{-6}	3.82×10^{-6}
256	256	cpu	1.95×10^{-7}	2.90×10^{-7}	5.32×10^{-7}	9.38×10^{-7}	1.81×10^{-6}	3.46×10^{-6}
256	256	Harley seal	5.08×10^{-7}	6.06×10^{-7}	7.85×10^{-7}	1.16×10^{-6}	1.14×10^{-6}	1.85×10^{-6}
256	256	Lookup	2.24×10^{-7}	3.13×10^{-7}	4.65×10^{-7}	5.71×10^{-7}	9.98×10^{-7}	1.83×10^{-6}
256	256	Lookup orig.	2.15×10^{-7}	2.90×10^{-7}	4.56×10^{-7}	7.82×10^{-7}	1.44×10^{-6}	2.75×10^{-6}
256	64	Popcnt movdq	1.60×10^{-7}	2.58×10^{-7}	4.73×10^{-7}	8.82×10^{-7}	1.73×10^{-6}	3.36×10^{-6}
256	64	Popcnt un. err.	1.86×10^{-7}	3.08×10^{-7}	5.42×10^{-7}	1.04×10^{-6}	2.02×10^{-6}	3.93×10^{-6}

3. SIMD implementation

This section covers the explicit vectorization of the operations presented in the previous section, using 256-bit and 512-bit AVX Intrinsics from the *AVX2* and *AVX512BW* extensions. This section also addresses several optimizations introduced both to the individual operations and the general exhaustive algorithm to improve the performance of the vectorized codes.

The *AVX2* vector extension was first introduced with the Intel Haswell microarchitecture (2013) while the *AVX512BW* extension first appeared in the Skylake-X processors (2017) of the Skylake microarchitecture. In this work, these two vector extensions are used not only to optimize the runtime of the epistasis detection tool on a long list of CPUs, but also to compare the performance that the two vector widths offer.

3.1. Vectorization of the genotype table calculation

The function `combine_subtable` shown in Listing A.1 is the one implementing the computation of a genotype subtable from two previous subtables, and thus our target for vectorization. In this function, we can identify a vectorization opportunity at the innermost loop, where the intersection of two rows from two tables is calculated by performing as many bitwise *AND* operations as values contained in the row (Line 9). This operation is already exploiting the data-parallelism that 64-bit operations offer, as the information of a singular individual is stored in a single bit of the data type `uint64_t`. With the introduction of 256 and 512-bit AVX instructions, the throughput of this operation can be multiplied.

Listing A.4 shows the implementation using 256-bit AVX Intrinsics from *AVX2*. For simplicity, we assume that the number of bytes in a row of the genotype table is divisible by the vector unit width. This is achieved by padding the rows with zeros if the number of individuals is not divisible by the width of the vector unit, and it will not influence the result of the following *popcount* operation. The new figure replaces the C++ code corresponding to the two array accesses, the *AND* and the store operations with AVX loads, *AND*s and store intrinsics. With just the introduction of the AVX Intrinsics, there is a front-end bound problem in which the CPU wastes many clock cycles waiting for instructions to be fetched. Therefore, to correct this behavior, the middle loop was unrolled completely so that the three rows from the second genotype table are processed concurrently.

The 512-bit vector implementation using intrinsics from the *AVX512BW* extension is almost identical to the one shown in Listing A.4, and thus it was omitted. The only differences are the name of the functions that implement the same operations for a 512-bit width, the types that these operations use and the step of the innermost loop, which doubles the one used in the 256-bit implementation.

3.2. Vectorization of the contingency table calculation

The main difference between the codes for calculating genotype and contingency tables (Listings A.1 and A.2, respectively) is the presence of the *popcount* operation. Up until very recently, with the introduction of the Intel Ice Lake processors, there was no AVX vector instruction implementing a *popcount*. Muła et al., in [25], have already explored this problem and they proposed multiple algorithms for implementing population counts using the *AVX2* extension. Furthermore, in their Github repository [26], they have developed updated versions of the algorithms to make use of the more recent *AVX512BW* and *AVX512VBMI* extensions.

Deciding which algorithm runs the fastest is not trivial and cannot be measured in isolation, as interleaving additional loads and bitwise AND operations in between *popcounts* will undoubtedly affect the performance of the function as a whole. For this reason, we implemented multiple versions of the `combine_and_popcount` function and compared the performance of each choice. Table 1 includes the elapsed time during the computation of a contingency table for the different implementations of the function on an Intel Xeon Gold 6240 CPU, the processor used for the experimental evaluation in Section 4, and compiled with GCC version 8.3.0. The table considers:

1. Two different vector widths for the bitwise AND operations: 256 and 512 bits.
2. Three different vector widths for the *popcount* operations: 64 bits, using the hardware *popcount* instruction from the Bit Manipulation Instructions (BMI) extension, and 256 and 512 bits, using the software implementations proposed in [25,26].
3. Six different table row widths: 256, 512, 1024, 2048, 4096 and 8192 individuals in each row (or 32, 64, 128, 256, 512 and 1024 bytes per row, respectively), equal for cases and controls.

From these results we can conclude that the fastest implementation is dependent on the width of the genotype tables. For less than 512 individuals per subtable, the best times are obtained by the implementations that make use of the 64-bit hardware *popcount* instruction. However, if we have more than 512 individuals, the lookup implementations for both the *AVX2* and *AVX512BW* extensions offer the fastest alternative for most of the widths tested. Taking a look at all of the epistasis studies referenced

throughout this paper, we can find that most of them consider a number of individuals between 512 and 4096. Therefore, for this work we will use the *AVX2* and *AVX512BW* implementations of the *popcount* lookup algorithm.

Vectorizing the `combine_and_popcount` function from Listing A.2 requires vectorizing its auxiliary subroutine `popcnt_subtable`. Starting with the *AVX2* implementation, Listing A.5 shows an implementation of the vectorized function, combining the computation of the new genotype table with the *popcount* lookup algorithm. This function includes the following modifications to the original lookup algorithm:

1. Instead of iterating over an input array as in the original popcount function (Lines 32–43 from file `popcnt-avx2-lookup.cpp` from [26]), `popcnt_subtable` consists of three nested loops: the two outer ones (Lines 35 and 36) combining the different rows of the input genotype tables, and the two innermost loops (Lines 38 and 74) applying the *popcount* iteration to each 256-bit word of the two selected rows. The first of the two innermost loops (Lines 38–67) maintains the original unrolling of eighth 256-bit words.

2. Each iteration step (inlined function `iter`) reads a 256-bit word from each table row (Lines 6 and 7), computes the bitwise `AND` of the two words (Line 8) and continues with the Muła *popcount* algorithm (Lines 9–15, which correspond to Figure 10 from [25]).

Listing A.6 shows the implementation of the same `popcnt_subtable` subroutine but using Intrinsics from the *AVX512BW* extension. The original *popcount* lookup algorithm for *AVX512BW* (file `popcnt-avx512bw-lookup.cpp` from [26]) is very similar to its *AVX2* implementation, with the obvious difference of not applying unrolling to its innermost loop (Lines 39–49). Therefore, the same considerations for the *AVX2* implementation of the function apply to the *AVX512BW* algorithm: the function combines the input genotype tables using three nested loops (Lines 14, 15 and 18), and each *popcount* iteration of the Muła algorithm is preceded by two loads that read a 512-bit word from each input genotype table (Lines 22 and 24) and a bitwise AND operation (Line 26).

3.3. Vectorization of the mutual information calculation

In contrast to the two previous functions, calculating the MI of a contingency table requires floating-point arithmetic, including multiplications, fused multiply-adds (FMAs) and logarithms. Multiplications and FMAs are supported natively, both for 256-bit and 512-bit vector operations, but there is no hardware instruction that implements a logarithm. However, Intel does provide an AVX logarithm routine through their Short Vector Math Library (SVML), an extension to the Intel Intrinsics available only with the Intel Compiler. GCC provides a vector implementation of the logarithm through GNU *libc*'s vector math library, available since version 2.22, although the number of vector functions available with GNU *libc* is much more limited compared to Intel's SVML.

Listing A.7 shows a C++ function implementing the MI computation using AVX Intrinsics from the *AVX2* extension. This code assumes that the contingency table size is divisible by the vector unit width. Similar to the genotype table and contingency table calculations, we can achieve this by padding the input contingency table with 0's, which will not contribute to the final MI value. The computation follows the same strategy of avoiding the calculations of the logarithm of zero as in the regular MI implementation (Listing A.3) but by different means: instead of skipping the logarithm altogether, which is not possible now unless all eight values of the vector are zero, the function replaces the zeros in the vector registers with ones that will evaluate to

zero and will not contribute in the following FMA operations (Lines 16–20, 25–29 and 33–36).

Moving onto *AVX512BW*, this extension provides mask registers and masked operations, which allow for the execution of operations only on some of the values contained in the vector register. Masked logarithms are a very convenient operation to skip the computation of the logarithm of zero. With masks we can avoid the zeros without having to blend two vector registers beforehand. Unfortunately, masked logarithms are only available under the SVML and for a vector width of 512 bits. The rest of the implementations still have to rely on the sequence of blends and logarithms.

Listing A.8 shows the same code implemented using 512-bit intrinsics from the *AVX512BW* extension. Comparisons are now made using the new intrinsic functions operating with 16-bit masks (Lines 18, 28 and 36) instead of a whole vector register, and the blend operation takes these masks as an argument. If the SVML is available, the logarithm plus blend sequences of operations (Lines 21, 31 and 38) can be replaced with a single call to the intrinsic `mm512_mask_log_ps`, which only calculates the logarithm on the positions specified by the mask.

The *AVX512BW* extension also includes mask functions for 256-bit operations. Therefore, for comparison purposes, a third version of the MI function using a width of 256 bits was also created. This version uses the same sequence of blend plus logarithm intrinsic functions shown in Listing A.8 both for the SVML and GNU's *libc* libraries since Intel does not include in the SVML a masked version of the 256-bit logarithm intrinsic.

3.4. Segmented exploration algorithm

Although the algorithm presented in Algorithm 1 could directly incorporate the SIMD functions described in the previous subsections, the performance per core would be penalized due to the interleaved execution of vector instructions running at very different frequencies. Intel CPUs, such as the Xeon Gold 6240 used during the evaluation, are known to downscale their CPU clock frequency based on the number of active cores and the sequence of instructions executed due to differences in power consumption and/or heat dissipation. In the processor technical document [27], Intel identifies three different frequency licenses in which the processor operates: non-AVX, AVX 2.0 and AVX-512 base core frequencies. Furthermore, different operations inside each license are not guaranteed to run at the same frequency, these are only base frequencies that the processor is guaranteed to run at. For example, floating-point arithmetic vector operations run at a slower clock frequency than integer arithmetic or bitwise vector operations.

As a direct consequence of this, the exploration algorithm would run on the lowest frequency imposed by any of the vector operations, since the change in frequency is not immediate and depends on the pipeline of operations executed. To resolve it, Algorithm 2 proposes a modification to the algorithm, segmenting the different operations into blocks of operations corresponding to similar frequency levels and therefore avoiding the frequency change problem.

Instead of declaring a single contingency table, the function now reserves space to store *b* combinations and compute *b* tables before applying MI to any of them (Lines 2–3). Combinations are now explored using two nested *while* loops, the outer one iterating until the stack is empty and all combinations starting with the SNP *i* have been explored (Line 8), and the inner one iterating until the block of *b* contingency tables has been filled (Line 10).

Every iteration of the innermost loop starts by checking if the stack is empty. If that is the case, and there are no more SNPs to

---

Algorithm 2: Segmented exploration

Input:
d: Data set containing n genotype tables, each representing a single SNP for m individuals
o: Order of the interactions to identify
b: Size of the block of operations

Output:
v: Vector of combinations of o SNPs, and their associated MI value

1   Allocate an array g of $o-2$ genotype tables, for sizes between 1 and $o-1$
2   Allocate an array a of b combinations of size o
3   Allocate an array c of b contingency tables of size o
4   Create an empty stack s
5   Calculate the marginal entropy $H(Y)$
6   $inv\_inds \leftarrow 1/m$
7   $i \leftarrow 0$
8   **while** *s is not empty* **or** *i < n* **do**
9     $l \leftarrow 0$
10   **while** $l < b$ **do**
11   **if** *s is empty* **then**
12   **if** $i \geq n$ **then**
13   Break from the inner while loop
      else
14        $g[1] \leftarrow d[i]$
15   **for** $j \leftarrow i+1$ **to** n **do**
16   Push the pair $\{i, j\}$ into the stack s
       end
17        $i \leftarrow i+1$
18   Continue on the next iteration of the loop
      end
     end
19   Pop the combination $\{c_1, \ldots, c_k\}$ from the top of the stack s
20   **if** $k < o$ **then**
21       $g[k] \leftarrow combine(g[k-1], d[c_k])$
22   **for** $j \leftarrow c_k + 1$ **to** n **do**
23   Push the new combination $\{c_1, \ldots, c_k, j\}$ into the stack s
      end
     else
24       $a[n] \leftarrow \{c_1, \ldots, c_k\}$
25       $c[n] \leftarrow combine\_and\_popcnt(\ g[o-1], d[c_k])$
26       $l \leftarrow l+1$
     end
   end
27   **for** $j \leftarrow 0$ **to** l **do**
28      $mi\_val \leftarrow MI(c[j], H(Y), inv\_inds)$
29   Add $a[j]$ and mi_val to the vector of results v
   end
  end

---

explore, the loop exits (Line 13); otherwise, the genotype table of the SNP i is copied in $g[1]$, all combinations of two SNPs starting with i are pushed into the stack, the counter i is increased and the execution continues on the next iteration of the inner while loop (Lines 14–18). If the stack is not empty, the loop operates in a similar manner as the old one: the top combination $\{c_1, \ldots, c_k\}$ of the stack is popped (Line 19). If k is smaller than the target interaction order o, its genotype table is computed and stored in the array of genotype tables g, and all subsequent combinations starting with $\{c_1, \ldots, c_k\}$ are pushed into the stack (Lines 21–23). Otherwise, $\{c_1, \ldots, c_k\}$ and its contingency table are stored in the a and c arrays, respectively (Lines 24–26). When the arrays a and c of b index combinations and tables, respectively, has been filled, the inner *while* finishes and a *for* loop iterates over all the computed contingency tables, calculating its MI and adding the combination into the vector of results v (Lines 27–29).

The selection of a proper value for the block size b is key in order to obtain good performance. It has to be large enough to make the impact of the transition between frequencies negligible, but not large enough to exceed the second-level cache of the processor. Through experimental testing, we found that an appropriate b for the Intel Xeon Gold 6240, the processor used in the evaluation, is $1474560/3^\circ$, with o being the order of the search. This size corresponds to the smallest block size tested at which the average running frequencies of the functions is very close or equal to the running frequency of these same functions in isolation.

4. Evaluation

We have conducted an extensive evaluation of the performance achieved by the automatic vectorization offered by the GCC and Intel compilers, in contrast to manual vectorization using Intel Intrinsics, when implementing a SIMD epistasis detection algorithm. It considers the performance of the different functions that compose the epistasis search in isolation, as well as the whole depth-first search algorithm. This evaluation starts by assaying the individual functions separately, and identifying which of the implementations obtains the best performance in each part. Then, the search algorithm is evaluated showcasing how the relative differences in time spent in each of the functions, and the operations that each function involves, influence the performance of the whole search. At last, the best performing implementation is compared against the original MPI3SNP [16] program using the compiler's automatic vectorization, to put into perspective the performance gain achieved.

Given the exponential time complexity of the operations that compose an epistasis search, and the search itself, it is difficult to represent elapsed time results for different problem sizes in the same graph and extract conclusions from them. For this reason, this evaluation uses the average elapsed time per cell or row (depending on the computation being evaluated) as the metrics to present the results. These measures of time express the compute time relative to the complexity of the computation, thus removing the impact of this complexity from the results and highlighting the differences in performance from multiple implementations of the same operation.

The two compilers used throughout the evaluation are the GNU C Compiler 8.3.0 (with the GNU *libc* version 2.29) and the Intel C++ Compiler 2020 (version 19.1.1.217). The same optimization flags were used for both compilers: -O3, -march= native and -mtune=native. Additionally, we enabled optimizations on floating-point arithmetic operations using -fast- math and -fp-model=fast for the two compilers respectively, as it is a requirement for GCC in order to vectorize some calls to the math library. Furthermore, for the automatic vectorization, we considered the effects of indicating a preference for a particular vector width during the compilation through the flags -mprefer-vector-width={256,512} for the GCC compiler and -qopt-zmm-usage={low,high} for the Intel compiler. Only the flag -qopt-zmm-usage=high had a positive impact on performance, thus it is the only one included in the results.

All experiments were run on an Intel Xeon Gold 6240, an 18-core CPU that implements the *AVX2, AVX512F, AVX512CD, AVX512BW, AVX512DQ, AVX512VL* and *AVX512VNNI* vector extensions. As mentioned in Section 3.4, performance during SIMD operation in modern Intel CPUs is tied to the number of active threads and the type of vector operations used in the instruction pipeline. This is stated in the processor technical document [27],
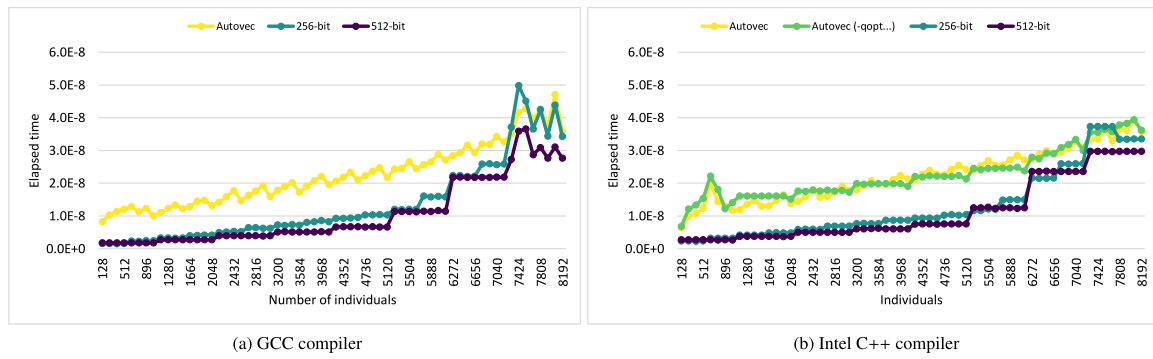
(a) GCC compiler
(b) Intel C++ compiler

Fig. 3. Average elapsed time per row during the calculation of genotype tables, for an increasing number of individuals and a fixed combination size of three, both using the GNU C Compiler and the Intel C++ Compiler.
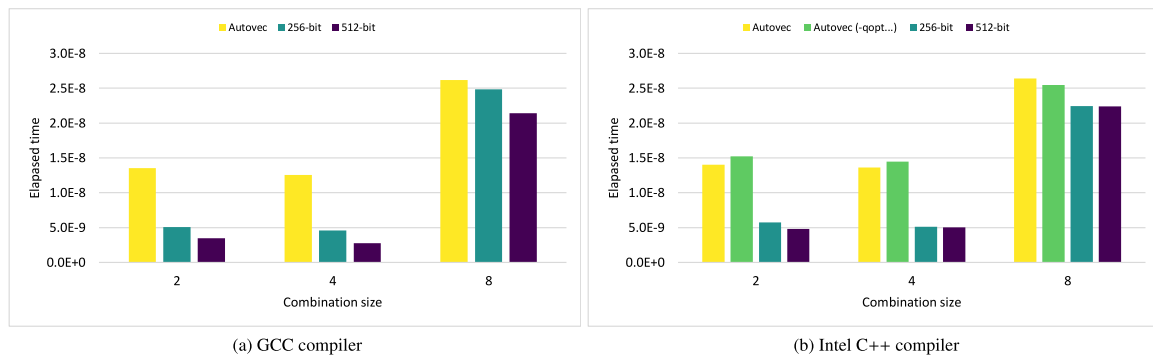

(a) GCC compiler
(b) Intel C++ compiler

Fig. 4. Average elapsed time per row during the calculation of genotype tables, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GNU C Compiler and the Intel C++ Compiler.

where the maximum core frequencies in turbo mode are specified attending to the number of cores and type of vector operation used. Therefore, to obtain a realistic multithreaded performance, elapsed times throughout the evaluation are measured during a simultaneous execution of the function in question on every core of the processor. The 18 different times are then averaged and presented as a single value.

4.1. Genotype table calculation performance

Figs. 3 and 4 represent the performance results for the genotype table calculation function. The figures compare the performance of the explicit vectorization using 256-bit and 512-bit vector instructions with the automatically vectorized code, both for the GCC and Intel compilers. The measure of time used in both figures is the average time per row, that is, the average elapsed time during the calculation of a single row of the table including both cases and controls, for all of the genotype tables of the order and number of individuals specified.

Both compilers are capable of automatically vectorizing this function with no problems. Despite this, and as the figures show, the performance of the autovectorization for both compilers is inferior than the performance of both explicitly vectorized alternatives.

Fig. 3 represents the time per row during the computation of genotype tables corresponding to a combination of three SNPs, for a growing number of individuals. The time per row grows linearly with the number of individuals, as every row of the genotype table contains information about all the individuals in the data. Both compilers show a gap between the performance of the automatic and explicit vectorizations that is present until a number of individuals higher than 7040. The 512-bit explicit implementation performs slightly better in general than the 256-bit one.

Fig. 4 represents the time per row during the computation of genotype tables corresponding to combinations of 2, 4 and 8 SNPs, using a fixed number of individuals of 2048. Here, the time per row should remain constant when increasing the size of the combinations (s), as the number of rows in the table (3^s) grows with the number of SNPs in combination considered, but each row contains the same 2048 individuals. This is the case for combination sizes smaller than eight, where the time per row remains mostly constant between combination sizes of two to seven. Starting at genotype tables of eight SNPs, there is an increase in the elapsed time due to the size of the operands and result tables exceeding the level 1 data cache of the processor, which is manifested in the results by bringing the vector and non-vector performances much closer.

For second and fourth-order interactions, both explicit vectorization alternatives obtain again better results than the vectorization applied by the compiler, with the 512-bit implementation performing the best. For eighth order interactions the performance gap is smaller, with less relative difference between implementations.

When taking a look at the frequencies at which the different implementations run, we observe that the genotype table calculation runs at 3270 MHz for the 256-bit vector width and 2805 MHz for the 512-bit vector width. In a different architecture, or in future Intel microarchitectures, where the difference in frequencies between vector widths could be smaller or nonexistent, we can expect the performance gap between the two widths to be larger. As an example, the elapsed time per row of calculating a fourth-order genotype table of 2048 individuals at a fixed frequency of 2.6 GHz (the base frequency of the processor) is 6.52×10^{-9} s and 3.00×10^{-9} s for the explicit 256-bit and 512-bit implementations under GCC, respectively; and 6.32×10^{-9} s and 4.04×10^{-9} s for the same implementations
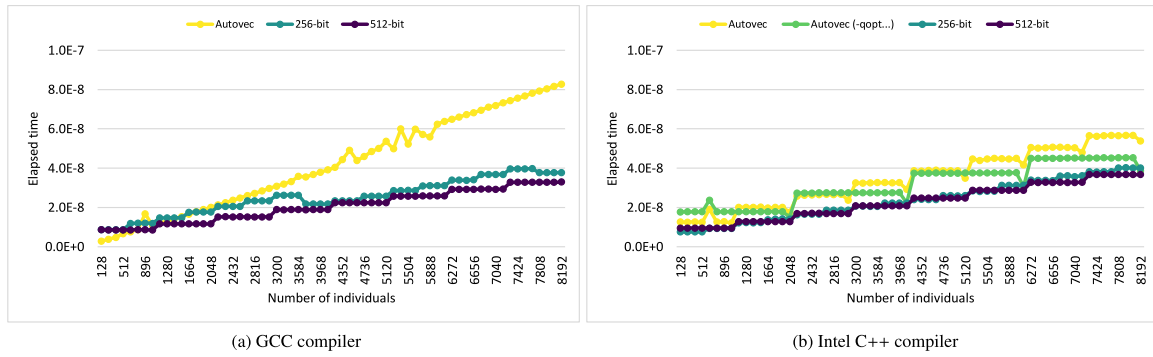
(a) GCC compiler

(b) Intel C++ compiler

Fig. 5. Average elapsed time per cell during the calculation of contingency tables, for an increasing number of individuals and a fixed combination size of three, both using the GNU C Compiler and the Intel C++ Compiler.
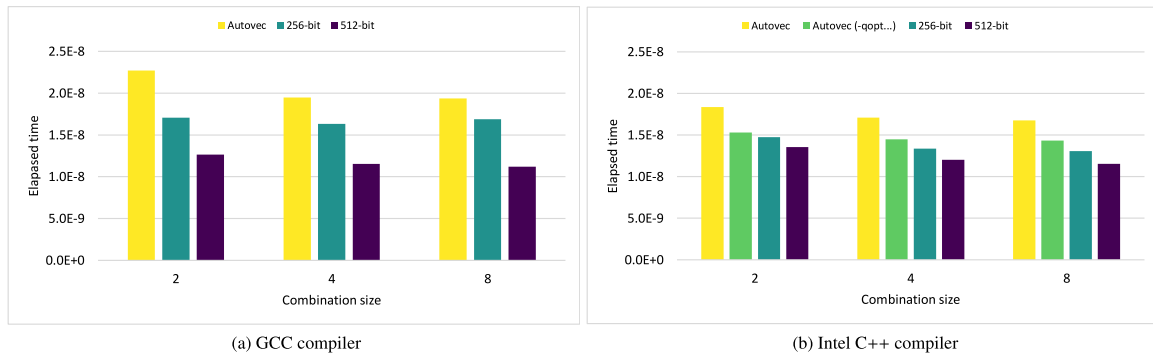

(a) GCC compiler

(b) Intel C++ compiler

Fig. 6. Average elapsed time per cell during the calculation of contingency tables, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GNU C Compiler and the Intel C++ Compiler.

under Intel, respectively. That is, the 512-bit implementation is 2.18 and 1.56 times faster than the 256-bit one for each compiler, significantly larger than what we observe in Fig. 4 between these two implementations (1.66 and 1.01).

4.2. Contingency table calculation performance

Figs. 5 and 6 represent the performance results for the contingency table calculation function. Similar to the figures from the genotype table calculation, these also compare the performance of the explicit vectorization using 256-bit and 512-bit vector operations with the automatically vectorized code using both GCC and Intel compilers. In this case, we use the average elapsed time per cell to represent the performance results, that is, the average time for the calculation of a single cell of the table, for all of the contingency tables of the order and number of individuals specified.

For this function, only the Intel compiler is capable of vectorizing the *popcount* operation via the introduction of its own vector implementation. GCC, on the other hand, refuses to vectorize this function due to the presence of the aforementioned operation inside the innermost loop.

Fig. 5 represents the time per cell during the computation of contingency tables corresponding to a combination of three SNPs, for a growing number of individuals. The elapsed time per cell during the creation of contingency tables also grows linearly with the number of individuals, since the function operates with the rows from the two previous genotype tables, which include the data of all individuals. The differences in compiler behavior are apparent: GCC results display a linear increase of the elapsed time per cell with the number of individuals, at a faster pace than the Intel results due to the lack of vectorization. It is also worth noting that there is a small reduction of the elapsed time per cell

in the explicit 256-bit vectorization under GCC at 3712 individuals, which corresponds to the minimum number of individuals required to enter the loop that includes unrolling (Listing A.5, Line 38). Anyhow, both explicit implementations are faster than the codes that the two compilers offer, with the 512-bit explicit vectorization being the fastest alternative.

Fig. 6 represents the time per cell during the computation of contingency tables for combination sizes of 2, 4 and 8, and using the same number of 2048 individuals. Analogous to the elapsed time per row during the calculation of genotype tables, the time per cell should also remain constant with the size of the combinations explored. However, contrary to those results, there is no increase in the elapsed time for eight order tables due to cache problems thanks to the avoidance of the genotype table storage. This is due to the merge of the last level genotype table calculations and contingency table computations in a single function. Results show that the explicit implementations are faster than the compiler-generated code, with the 512-bit vectorization being the fastest implementation.

Similar to the genotype table calculations, we observe that the contingency table computation function runs at a frequency of 3195 MHz for the 256-bit vector width and 2800 MHz for the 512-bit vector width. If we run the function at a fixed frequency of 2.6 GHz (the base frequency of the processor), the elapsed time per cell of calculating a fourth-order contingency table of 2048 individuals is 1.97×10^{-8}s and 1.24×10^{-8}s for the explicit 256-bit and 512-bit implementations under GCC, respectively; and 1.58×10^{-8}s and 1.29×10^{-8}s for the same implementations under Intel, respectively. This represents a speedup of 1.59 and 1.23 between vector widths for each compiler, slightly larger than those observed in Fig. 6 (1.42 and 1.11).
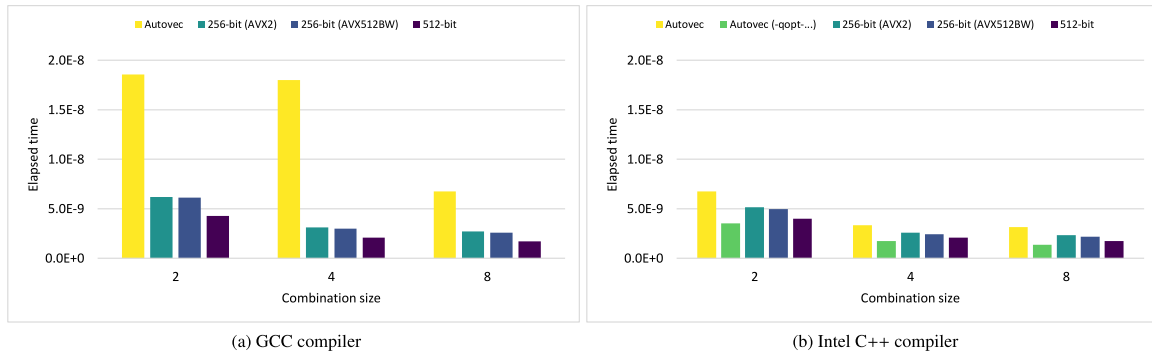
(a) GCC compiler

(b) Intel C++ compiler

Fig. 7. Average elapsed time per cell during the calculation of the MI metric, for combination sizes of 2, 4 and 8, both using the GNU C Compiler and the Intel C++ Compiler.

4.3. Mutual information calculation performance

Fig. 7 shows the performance results for the MI computation function. This figure compares the performance of the automatically vectorized code with three explicit implementations: two 256-bit vector implementations using Intrinsics from the *AVX2* and *AVX512BW* extensions, respectively, and a 512-bit vector implementation using Intrinsics from the *AVX512BW* extension. Here we also use the time per cell to represent the performance results. This time measures the average elapsed time during the MI calculations corresponding to a single cell of the contingency table (both for cases and controls), for all of the tables of the specified order. The number of cells of a contingency table only depends on the number of SNPs in combination. Thus, MI, as opposed to the previous routines, does not depend on the number of individuals.

Results show that the time per cell for the explicit vector implementations generally decreases with the table size, despite the fact that, ideally, the workload per contingency table cell should remain constant regardless of the size of the table. This can mostly be attributed to the additional computations derived from the padding introduced in the input contingency tables. The larger the tables are, the lower the number of unnecessary computed cells is relative to the total number of cells in the table.

The best vector performance is achieved by the automatic vectorization of the Intel Compiler when coupled with the flag `-qopt-zmm-usage=high`. In contrast, GCC's automatic vectorization does not vectorize the loop, despite having a vectorized logarithm function available in the GNU *libc* math library. Explicit vectorization using 512-bit AVX instructions obtains the best performance out of the explicit vector implementations (for GCC it is the fastest alternative). Furthermore, the introduction of 256-bit *AVX512BW* instructions in the function have no significant impact on the elapsed time when compared to the *AVX2* implementation.

When examining the assembly code to characterize the difference in performance between the explicit vector implementations and the code that the Intel auto-vectorizer generates, we found that the Intel Compiler calls a function from the SVML that is not available using Intrinsics: `__svml_logf8_mask_e9` (a logarithm function for a vector width of 256 bits that uses a masked input). Therefore, in some scenarios, explicit vectorization may never obtain a performance equal or better than Intel's auto-vectorization due to the difference in SVML functions available through Intrinsics.

As for the frequencies at which the function is executed, the 256-bit implementation runs at 2805 MHz (we only measured the 256-bit implementation using *AVX2* Intrinsics, since the elapsed times are almost the same) while the 512-bit implementation runs at 2500 MHz. These frequencies are considerably lower than the two previous functions due to the usage of floating-point arithmetic operations. If the running frequencies are fixed to 2.5 GHz (slightly lower than the base frequency because the 512-bit implementation runs at this frequency), we observe that the elapsed time per cell of calculating the MI of a fourth-order contingency table are 3.48×10^{-9} s and 2.10×10^{-9} s for the 256-bit and 512-bit vectorizations under GCC, respectively, and 2.87×10^{-9} s and 2.10×10^{-9} s under the Intel Compiler. This represents a speedup of 1.66 and 1.36 between vector widths for each compiler, respectively, slightly larger than those observed in Fig. 7 (1.49 and 1.23).

4.4. Exhaustive search performance

At last, Figs. 8 and 9 present the performance results for the whole exhaustive search algorithm. The two figures compare the performance of the 256 and 512-bit explicit vectorization approaches using operations from the *AVX2* and *AVX512BW* extensions, with the automatically vectorized code using both GCC and Intel compilers, for both versions of the search algorithm presented in Algorithms 1 and 2.

Figs. 8 and 9 use the average elapsed time per cell to represent the performance results. The time per cell is the average elapsed time spent during the computation of a single contingency table cell, the subsequent MI operations corresponding to the cell of the table and a fraction of the time spent during the calculations of previous genotype tables (this time is equally divided across all cells of all combinations that make use of that genotype table), for all of the contingency tables of the order, number of SNPs and individuals specified.

Fig. 8 represents the time per cell, shown as stacked bars indicating the fraction of the time spent in each of the functions, during the search of epistasis in combinations of 2, 4 and 8 SNPs, and for a fixed number of individuals of 2048. The number of SNPs was tied to the size of the combinations so that the workload among different explorations was as similar as possible. Table 2 indicates, for each exploration order, the number of SNPs selected, the resulting number of SNP combinations of said order, the number of different cells among those combinations and the difference in workload that exploration order and number of SNPs supposes from the first one. The figure shows that the 512-bit explicit vectorization performs the best out of all of the versions compared, which is coherent with what we saw during the evaluation of the individual functions. The 256-bit explicit implementations obtain practically the same results and the only compiler-generated vectorization that beats any of the explicit vectorizations is the Intel Compiler when coupled with the optimization flag `-qopt-zmm-usage=high` for low-order epistasis searches.
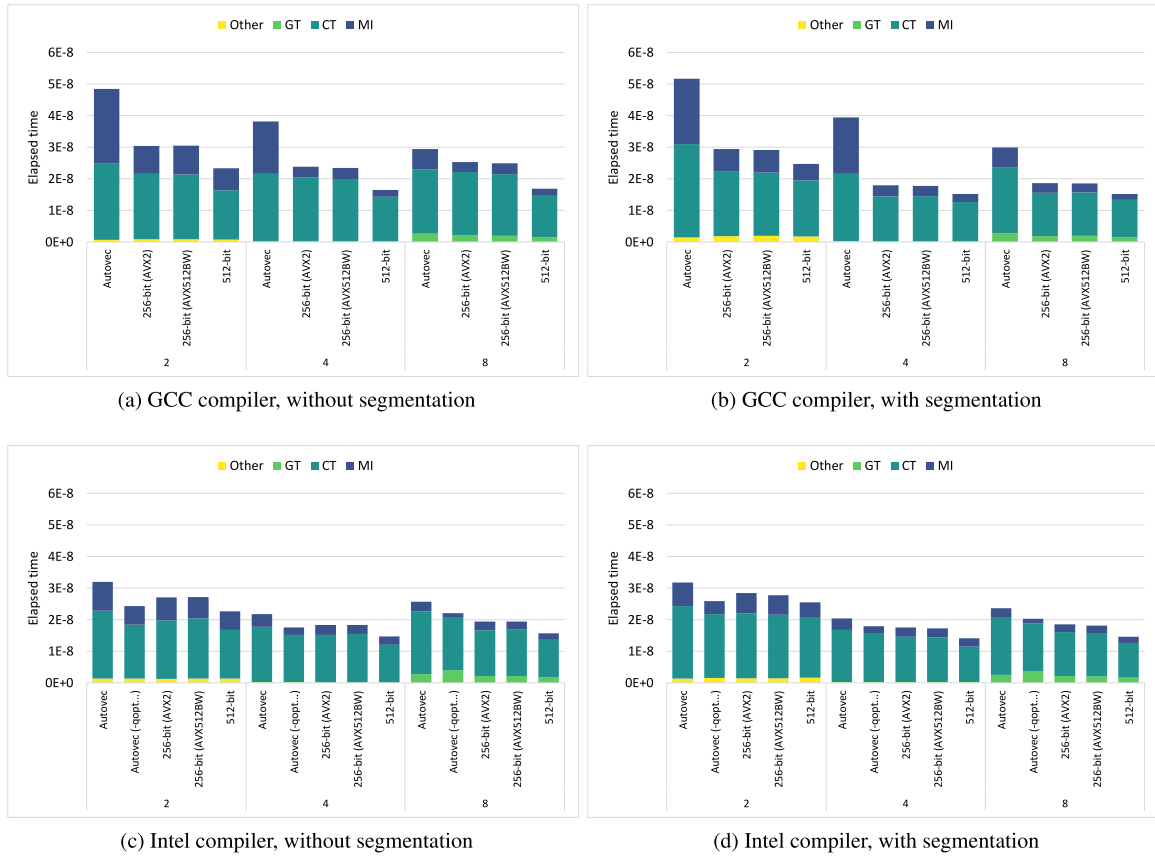
(a) GCC compiler, without segmentation

(b) GCC compiler, with segmentation

(c) Intel compiler, without segmentation

(d) Intel compiler, with segmentation

Fig. 8. Average elapsed time per cell during the exhaustive search of epistasis, for combination sizes of 2, 4 and 8 and a fixed number of 2048 individuals, both using the GCC and Intel compilers. The times for each approach is divided into the calculation of the Genotype Tables (GT), Contingency Tables (CT), Mutual Information (MI) and the rest of the operations included in the algorithm.

(a) GCC compiler, without segmentation

(b) GCC compiler, with segmentation

(c) Intel compiler, without segmentation

(d) Intel compiler, with segmentation
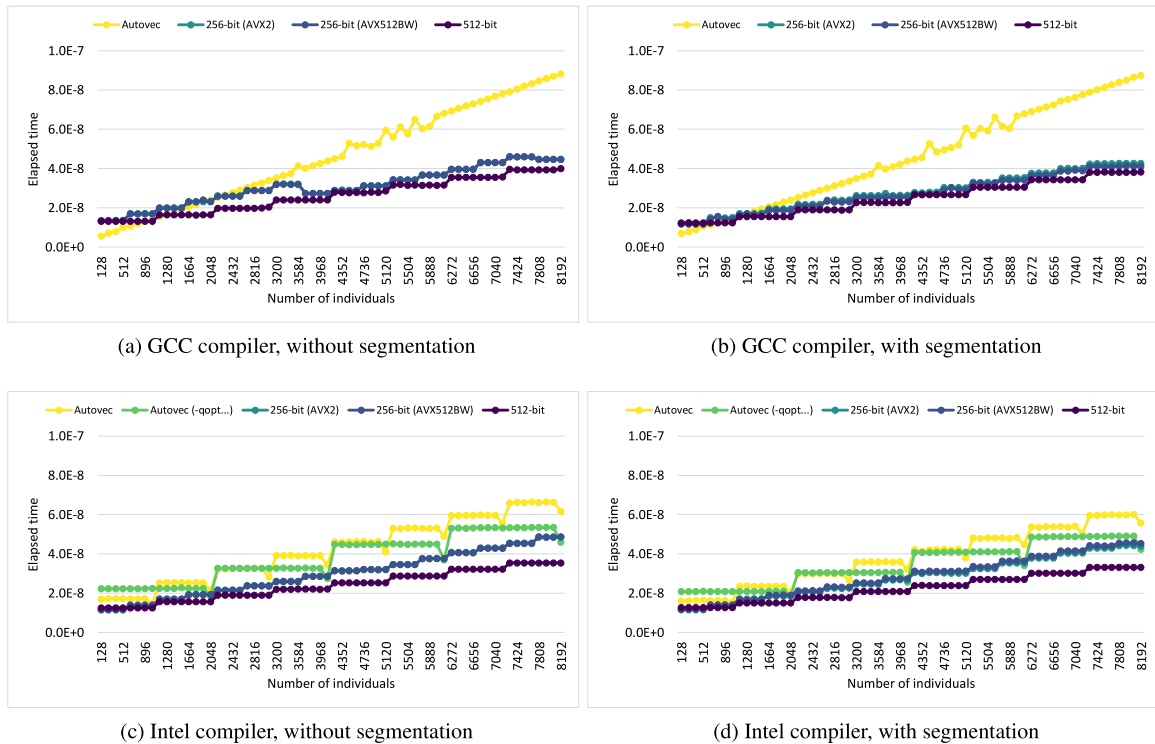
Fig. 9. Average elapsed time per cell during the exhaustive search of epistasis, for an increasing number of individuals, a fixed number of 680 SNPs, a fixed combination size of three and for both the GCC and Intel compilers.

Table 2

Combination size and number of SNPs used during the exhaustive search evaluation, indicating the resulting number of combinations and table cells, and the relative workload deviation. The total number of cells is the product between the number of combinations (as indicated by the previous column) and the number of cells in each contingency table, 3^{size}, for a particular combination size. The deviation is the difference between the total number of cells for a particular combination size and SNP number, and the number of cells for combinations of two SNPs and 25 000 SNPs, relative to the latter one.

Size	SNPs	Combinations	Total cells	Deviation
2	25 000	312 487 500	2 812 387 500	+0.00%
4	242	34 389 810	2 785 574 610	−0.95%
8	23	490 314	3 216 950 154	+14.39%

Table 3

Elapsed time, in seconds, and speedup of the explicitly vectorized 512-bit implementation compared to MPI3SNP. Results are the average of three executions of a single-threaded third-order search.

| | SNPs | Inds. | MPI3SNP | 512-bit vectorization | |
			Runtime	Runtime	Speedup
GCC	1000	1000	200.77	39.09	5.14
	1000	2000	254.43	50.97	4.99
	4000	1000	25 698.86	2527.58	10.17
	4000	2000	31 506.60	3401.53	9.26
Intel	1000	1000	391.47	40.64	9.63
	1000	2000	735.52	48.98	15.02
	4000	1000	25 113.33	2626.10	9.56
	4000	2000	47 179.35	3288.37	14.35

The segmentation of operations introduced in the algorithm has an overall positive effect on the explicitly vectorized implementations. From a CPU frequency perspective, the segmentation algorithm achieves its goal. When there is no separation between integer and binary arithmetic, and floating-point arithmetic, the whole program runs at 2.8 and 2.5 GHz for the 256-bit and 512-bit implementations, respectively. However, when there is segmentation, genotype and contingency table calculations run at 3.05 and 2.75 GHz, and the MI operations run at 2.8 and 2.5 GHz for each implementation respectively. From the performance perspective, the segmentation strategy only works with the explicitly vectorized implementations and results in a noticeable reduction of the elapsed time of the searches under GCC, and a much smaller gain under the Intel Compiler for high-order interactions.

Fig. 9 represents the time per cell for a growing number of individuals from 128 to 8192, using a fixed combination size of three and a fixed number of SNPs of 680. Although the number of individuals is irrelevant to the calculation of the MI, it affects the calculation of the genotype tables and contingency tables, and therefore the time per cell during the whole search also grows linearly with the number of individuals, although with a less pronounced slope than the two first individual operations. These two subfigures show similar behavior to the one shown in Fig. 5 because the calculation of the contingency table accounts for the majority of the elapsed time during the whole search.

Results from Fig. 9 show that the explicit vectorization using 512-bit operations achieve the best times. These are very similar to those of the contingency table calculation function (Fig. 5), which makes sense since it is the most time-consuming function of the algorithm as Fig. 8 showed. The best implementation is again the explicit 512-bit vectorization.

4.5. Performance of the vectorized search compared against MPI3SNP

To conclude the evaluation, Table 3 compares the elapsed time required to complete a third-order epistasis search using the original MPI3SNP [16] program and the explicit 512-bit vectorization proposed in this paper, for an input data consisting of 1000 and 4000 SNPs and 1000 and 2000 individuals, and using a single core of the processor. MPI3SNP was compiled using the same flags indicated during the introduction of this section, enabling for both compilers the automatic vectorization. Results show that the vector implementation of the algorithm speeds the execution up by an average factor of 7 using GCC and 12 using the Intel C++ Compiler. The speedup with the Intel Compiler is in part due to its poor memory handling when allocating memory for objects inside a loop, something that has been accounted for in the new implementation and that MPI3SNP does not do. Therefore, we believe that the speedups obtained by GCC paint a more realistic picture of what speedups should be expected of this algorithm.

5. Conclusions

Epistasis detection is still a biological problem that is far from being resolved. Despite the efforts, there is no single method that can be applied to a genome-wide scale data set to locate high order epistasis interactions reliably [12]. In this work, we propose different SIMD implementations that exploit the parallelization opportunities inherent to the epistasis detection problem in order to speed up the execution of an exhaustive search. This is achieved by the introduction of AVX Intrinsics functions during the calculation of the genotype tables, contingency tables and MI metric. We also include general optimization strategies, such as the segmentation of the operation pipeline due to the license-based downclocking on Intel processors, and other optimizations specific to this code, such as the loop unrolling during the calculation of genotype and contingency tables, or the avoidance of logarithms of 0 during the MI calculation. Although this work considers a specific exhaustive search algorithm, many of these vectorization and optimization techniques could be directly applied to a multitude of epistasis detection methods in the literature where genotype and contingency tables are constructed to assess the association between a genotype combination and a particular phenotype.

The results obtained highlight the potential of the SIMD parallelization when applied to the epistasis detection problem. For example, the runtime under GCC of an exhaustive search of an interaction consisting of three SNPs from two data sets containing 4000 SNPs and 1000 and 2000 individuals, respectively, was reduced from 428 and 525 min using MPI3SNP down to 42 and 57 min when using the 512-bit vector implementation proposed in this work. The observed speedups are not exclusive to single-core executions and will benefit multi-threaded runs, accelerating the computations in each of the CPU cores used.

The autovectorization provided by the compilers showed varying degrees of success attending to the compiler and the operations considered. Intel, for example, was capable of vectorizing all operations while GCC fell short. As for the performance achieved, we observed that optimization flags play a big role in the resulting performance of the code generated. GCC required the `-fast-math` flag to be capable of vectorizing calls to the math library, while Intel improved the performance significantly with the usage of the flag `-qopt-zmm-usage=high`. With respect to performance, Intel's autovectorization remained competitive with the explicit implementations for low-order interaction searches but fell behind when moving past fourth-order interactions. GCC's autovectorization, on the other hand, was never close to the performance of the explicit implementations due to its failure of vectorizing the operations.

Although the proposed explicit implementations are faster, they are tailored to the x86_64 CPU architecture. For different architectures, autovectorization is the only alternative at the

moment, and the results obtained in this study suggest that compilers will be able to vectorize some computations of the algorithm, if not all. If the vector width is comparable to those of AVX, as is the case with ARM's Scalable Vector Extension, this algorithm is likely to be effective. However, for architectures with much larger vector widths, e.g. the NEC SX-Aurora TSUBASA vector processor, this algorithm will be inadequate, as there are not enough values in the calculation of genotype and contingency tables to fill the whole vector register.

Moving forward, with future CPU microarchitectures and the introduction of new AVX extensions, it is reasonable to expect the performance of the SIMD epistasis detection algorithm to improve even further. During the evaluation we saw the effect that the Intel frequency model had on the performance attending to the width of the operations, penalizing the larger vector widths. If these differences in frequency are reduced in upcoming CPUs, the performance will consequently increase. Furthermore, with future AVX instructions, for example, the *popcount* operation from the *AVX512VPOPCNTDQ* extension recently introduced with Intel Skylake processors, some operations of the algorithm will allow for a more efficient implementation.

This work also presents some limitations and lines of future work:

1. The algorithm described here is single-thread, and only focuses on the SIMD performance of a sequential execution. Combining the proposed SIMD implementations with a multi-thread or multi-process execution is fundamental for exploiting all the computing power that current CPUs and clusters of CPUs offer.
2. This paper focuses solely on the x86_64 architecture, the most extended architecture in general-purpose computers and high-performance clusters. Therefore, the proposed algorithm may or may not be appropriate for different CPU architectures, GPUs or vector processors such as the aforementioned NEC SX-Aurora TSUBASA. Studying different implementations of this algorithm, or a different algorithm better suited to the characteristics of another architecture, is something to consider.

All codes presented in this work are available online.[1]

CRediT authorship contribution statement

Christian Ponte-Fernández: Conceptualization, Software, Investigation, Visualization, Writing – original draft. **Jorge González-Domínguez:** Conceptualization, Writing – review & editing, Supervision. **María J. Martín:** Conceptualization, Writing – review & editing, Supervision.

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.

Acknowledgments

Appendix. C++ functions

Listing A.1: Genotype table combination C++ function

```cpp
inline void combine_subtable(
  const uint64_t *gt_tbl1, const size_t size1,
  const uint64_t *gt_tbl2, const size_t words,
  uint64_t *gt_tbl3)
{
  for (size_t i = 0; i < size1; i++) {
    for (size_t j = 0; j < 3; j++) {
      for (size_t k = 0; k < words; k++) {
        gt_tbl3[(i * 3 + j) * words + k] =
            gt_tbl1[i * words + k] &
            gt_tbl2[j * words + k];
      }
    }
  }
}

void combine(
  const GenotypeTable<uint64_t> &t1,
  const GenotypeTable<uint64_t> &t2,
  GenotypeTable<uint64_t> &out)
{
  combine_subtable(t1.cases, t1.size, t2.cases,
    t1.cases_words, out.cases);
  combine_subtable(t1.ctrls, t1.size, t2.ctrls,
    t1.ctrls_words, out.ctrls);
}
```

Listing A.2: Contingency table calculation C++ function

```cpp
inline void popcnt_subtable(
  const uint64_t *gt_tbl1, const size_t size1,
  const uint64_t *gt_tbl2, const size_t words,
  uint32_t *ct_tbl, const size_t ct_size)
{
  for (size_t i = 0; i < size1; i++) {
    for (size_t j = 0; j < 3; j++) {
      ct_tbl[i * 3 + j] = 0;
      for (size_t k = 0; k < words; k++) {
        ct_tbl[i * 3 + j] += std::bitset<64>(
          gt_tbl1[i * words + k] &
          gt_tbl2[j * words + k]).count();
      }
    }
  }
}

void combine_and_popcnt(
    const GenotypeTable<uint64_t> &t1,
    const GenotypeTable<uint64_t> &t2,
    ContingencyTable<uint32_t> &out)
{
  popcnt_subtable(t1.cases, t1.size, t2.cases,
    t1.cases_words, out.cases, out.size);
  popcnt_subtable(t1.ctrls, t1.size, t2.ctrls,
    t1.ctrls_words, out.ctrls, out.size);
}
```

Listing A.3: MI computation C++ function

```
1  float MI(
2    const ContingencyTable<uint32_t> &table,
3    const float h_y,
4    const float iinds)
5  {
6    size_t i;
7    float h_x = 0.0f, h_all = 0.0f;
8    float p_case, p_ctrl, p_any;
9    const size_t table_size = table.size;
10   for (i = 0; i < table_size; i++) {
11     p_case = table.cases[i] * iinds;
12     if (p_case != 0.0f) {
13       h_all -= p_case * logf(p_case);
14     }
15     p_ctrl = table.ctrls[i] * iinds;
16     if (p_ctrl != 0.0f) {
17       h_all -= p_ctrl * logf(p_ctrl);
18     }
19     p_any = p_case + p_ctrl;
20     if (p_any != 0.0f) {
21       h_x -= p_any * logf(p_any);
22     }
23   }
24   return h_x + h_y - h_all;
25 }
```

Listing A.4: Genotype table calculation auxiliary C++ function vectorized with *AVX2* Intrinsics

```
1  inline void combine_subtable(
2    const uint64_t *gt_tbl1, const size_t size1,
3    const uint64_t *gt_tbl2, const size_t words,
4    uint64_t *gt_tbl3)
5  {
6    size_t i, j, k;
7    const __m256i *ptr1 = gt_tbl1;
8    for (i = 0; i < size1; ++i) {
9      const __m256i *ptr2_1 = gt_tbl2 + 0 * words;
10     const __m256i *ptr2_2 = gt_tbl2 + 1 * words;
11     const __m256i *ptr2_3 = gt_tbl2 + 2 * words;
12     __m256i *ptr3_1 = gt_tbl3 + (i*3+0) * words;
13     __m256i *ptr3_2 = gt_tbl3 + (i*3+1) * words;
14     __m256i *ptr3_3 = gt_tbl3 + (i*3+2) * words;
15     for (k = 0; k < words; k += 4) {
16       __m256i y0 = _mm256_load_si256(ptr1++);
17       __m256i y1 = _mm256_load_si256(ptr2_1++);
18       __m256i y2 = _mm256_load_si256(ptr2_2++);
19       __m256i y3 = _mm256_load_si256(ptr2_3++);
20       _mm256_store_si256(ptr3_1++,
21         _mm256_and_si256(y0, y1));
22       _mm256_store_si256(ptr3_2++,
23         _mm256_and_si256(y0, y2));
24       _mm256_store_si256(ptr3_3++,
25         _mm256_and_si256(y0, y3));
26     }
27   }
28 }
```

Listing A.5: Contingency table calculation auxiliary C++ function vectorized with *AVX2* Intrinsics

```
1  inline void iter(
2    const uint64_t *ptr1, const uint64_t *ptr2,
3    const __m256i &lu, const __m256i &low_mask,
4    __m256i &local)
5  {
6    __m256i o1 = _mm256_load_si256(ptr1);
7    __m256i o2 = _mm256_load_si256(ptr2);
8    __m256i vec = _mm256_and_si256(o1, o2);
9    __m256i lo = _mm256_and_si256(vec, low_mask);
10   __m256i hi = _mm256_and_si256(
11     _mm256_srli_epi16(vec, 4), low_mask);
12   __m256i popcnt1 = _mm256_shuffle_epi8(lu, lo);
13   __m256i popcnt2 = _mm256_shuffle_epi8(lu, hi);
14   local = _mm256_add_epi8(local, popcnt1);
15   local = _mm256_add_epi8(local, popcnt2);
```

```
16 }
17
18 inline void popcnt_subtable(
19   const uint64_t *gt_tbl1, const size_t size1,
20   const uint64_t *gt_tbl2, const size_t words,
21   uint32_t *ct_tbl, const size_t ct_size)
22 {
23   const __m256i lookup = _mm256_setr_epi8(
24     /* 0 */ 0, /* 1 */ 1, /* 2 */ 1, /* 3 */ 2,
25     /* 4 */ 1, /* 5 */ 2, /* 6 */ 2, /* 7 */ 3,
26     /* 8 */ 1, /* 9 */ 2, /* a */ 2, /* b */ 3,
27     /* c */ 2, /* d */ 3, /* e */ 3, /* f */ 4,
28     /* 0 */ 0, /* 1 */ 1, /* 2 */ 1, /* 3 */ 2,
29     /* 4 */ 1, /* 5 */ 2, /* 6 */ 2, /* 7 */ 3,
30     /* 8 */ 1, /* 9 */ 2, /* a */ 2, /* b */ 3,
31     /* c */ 2, /* d */ 3, /* e */ 3, /* f */ 4);
32   const __m256i low_mask = _mm256_set1_epi8(0xf);
33
34   size_t i, j, k;
35   for (i = 0; i < size1; ++i) {
36     for (j = 0; j < 3; ++j) {
37       __m256i acc = _mm256_setzero_si256();
38       for (k = 0; k + 32 <= words; k += 32) {
39         __m256i local = _mm256_setzero_si256();
40         iter(gt_tbl1 + i * words + k + 0,
41           gt_tbl2 + j * words + k + 0,
42           lookup, low_mask, local);
43         iter(gt_tbl1 + i * words + k + 4,
44           gt_tbl2 + j * words + k + 4,
45           lookup, low_mask, local);
46         iter(gt_tbl1 + i * words + k + 8,
47           gt_tbl2 + j * words + k + 8,
48           lookup, low_mask, local);
49         iter(gt_tbl1 + i * words + k + 12,
50           gt_tbl2 + j * words + k + 12,
51           lookup, low_mask, local);
52         iter(gt_tbl1 + i * words + k + 16,
53           gt_tbl2 + j * words + k + 16,
54           lookup, low_mask, local);
55         iter(gt_tbl1 + i * words + k + 20,
56           gt_tbl2 + j * words + k + 20,
57           lookup, low_mask, local);
58         iter(gt_tbl1 + i * words + k + 24,
59           gt_tbl2 + j * words + k + 24,
60           lookup, low_mask, local);
61         iter(gt_tbl1 + i * words + k + 28,
62           gt_tbl2 + j * words + k + 28,
63           lookup, low_mask, local);
64         acc = _mm256_add_epi64(acc,
65           _mm256_sad_epu8(local,
66             _mm256_setzero_si256()));
67       }
68
69       __m256i local = _mm256_setzero_si256();
70       for (; k < words; k += 4) {
71         iter(gt_tbl1 + i * words + k,
72           gt_tbl2 + j * words + k,
73           lookup, low_mask, local);
74       }
75       acc = _mm256_add_epi64(acc,
76         _mm256_sad_epu8(local,
77           _mm256_setzero_si256()));
78
79       ct_tbl[i * 3 + j] =
80         _mm256_extract_epi64(acc, 0) +
81         _mm256_extract_epi64(acc, 1) +
82         _mm256_extract_epi64(acc, 2) +
83         _mm256_extract_epi64(acc, 3);
84     }
85   }
86   for (i = size1 * 3; i < ct_size; ++i) {
87     ct_tbl[i] = 0;
88   }
89 }
```

Listing A.6: Contingency table calculation auxiliary C++ function vectorized with *AVX512BW* Intrinsics

```
1  inline void popcnt_subtable(
2    const uint64_t *gt_tbl1, const size_t size1,
```

```
3    const uint64_t *gt_tbl2, const size_t words,
4    uint32_t *ct_tbl, const size_t ct_size)
5  {
6    const __m512i lookup = _mm512_setr_epi64(
7      0x0302020102010100llu,0x0403030203020201llu,
8      0x0302020102010100llu,0x0403030203020201llu,
9      0x0302020102010100llu,0x0403030203020201llu,
10     0x0302020102010100llu,0x0403030203020201llu);
11   const __m512i low_mask = _mm512_set1_epi8(0xf);
12
13   size_t i, j, k, l;
14   for (i = 0; i < size1; ++i) {
15     for (j = 0; j < 3; ++j) {
16       k = 0;
17       __m512i acc = _mm512_setzero_si512();
18       while (k < words) {
19         __m512i local = _mm512_setzero_si512();
20         for (l = 0; l < 255 / 8 && k < words;
21              ++l, k += 8) {
22           __m512i z0 = _mm512_load_si512(
23             gt_tbl2 + j * words + k);
24           __m512i z1 = _mm512_load_si512(
25             gt_tbl1 + i * words + k);
26           __m512i z2 = _mm512_and_si512(z0, z1);
27           __m512i lo = _mm512_and_si512(
28             z2, low_mask);
29           __m512i hi = _mm512_and_si512(
30             _mm512_srli_epi32(z2, 4), low_mask);
31           __m512i popcnt1 = _mm512_shuffle_epi8(
32             lookup, lo);
33           __m512i popcnt2 = _mm512_shuffle_epi8(
34             lookup, hi);
35           local = _mm512_add_epi8(local,popcnt1);
36           local = _mm512_add_epi8(local,popcnt2);
37         }
38         acc = _mm512_add_epi64(acc,
39           _mm512_sad_epu8(local,
40             _mm512_setzero_si512()));
41       }
42       ct_tbl[i * 3 + j] =
43         _mm512_reduce_add_epi64(acc);
44     }
45   }
46   for (i = size1 * 3; i < ct_size; ++i) {
47     ct_tbl[i] = 0;
48   }
49 }
```

Listing A.7: MI computation C++ function vectorized with *AVX2* Intrinsics

```
1  float MI(
2    const ContingencyTable<uint32_t> &table,
3    const float h_y,
4    const float iinds)
5  {
6    const __m256 ones = _mm256_set1_ps(1.0);
7    const __m256 ii = _mm256_set1_ps(iinds);
8    __m256 h_x = _mm256_setzero_ps();
9    __m256 h_all = _mm256_setzero_ps();
10   __m256i y0;
11   __m256 y1, y2, y3, y4, y5;
12   for (auto i = 0; i < table.size; i += 8) {
13     y0 = _mm256_load_si256(table.cases + i);
14     y3 = _mm256_mul_ps(_mm256_cvtepi32_ps(y0), ii);
15     // Identify cells with 0's
16     y1 = _mm256_cmp_ps(y0,
17       _mm256_setzero_ps(), _CMP_NEQ_OQ);
18     // Replace 0's with 1's before log
19     y4 = _mm256_log_ps(_mm256_blendv_ps(
20       ones, y3, y1));
21     h_all = _mm256_fmadd_ps(y3, y4, h_all);
22     y0 = _mm256_load_si256(table.ctrls + i);
23     y4 = _mm256_mul_ps(_mm256_cvtepi32_ps(y0), ii);
24     // Identify cells with 0's
25     y2 = _mm256_cmp_ps(y0,
26       _mm256_setzero_ps(), _CMP_NEQ_OQ);
27     // Replace 0's with 1's before log
28     y5 = _mm256_log_ps(_mm256_blendv_ps(
29       ones, y4, y2));
```

```
30     h_all = _mm256_fmadd_ps(y4, y5, h_all);
31     y5 = _mm256_add_ps(y3, y4);
32     // Merge previous masks
33     y1 = _mm256_or_ps(y1, y2);
34     // Replace 0's with 1's before log
35     y3 = _mm256_log_ps(_mm256_blendv_ps(
36       ones, y5, y1));
37     h_x = _mm256_fmadd_ps(y5, y3, h_x);
38   }
39   y3 = _mm256_hadd_ps(h_all, h_x);
40   return (y3[0] + y3[1] + y3[4] + y3[5]) -
41     h_y - (y3[2] + y3[3] + y3[6] + y3[7]);
42 }
```

Listing A.8: MI computation C++ function vectorized with *AVX512BW* Intrinsics

```
1  float MI(
2    const ContingencyTable<uint32_t> &table,
3    const float h_y,
4    const float iinds)
5  {
6    const __m512 ones = _mm512_set1_ps(1.0);
7    const __m512 ii = _mm512_set1_ps(inv_inds);
8    __m512 h_x = _mm512_setzero_ps();
9    __m512 h_all = _mm512_setzero_ps();
10   __m512i z0;
11   __m512 z1, z2, z3, z4;
12   __mmask16 mask1, mask2, mask3;
13   for (auto i = 0; i < table.size; i += 8) {
14     z0 = _mm512_load_si512(table.cases + i);
15     z2 = _mm512_mul_ps(
16       _mm512_cvtepi32_ps(z0), ii);
17     // Identify cells with 0's
18     mask1 = _mm512_cmp_epi32_mask(z0,
19       _mm512_setzero_si512(), _MM_CMPINT_NE);
20     // Replace 0's with 1's before log
21     z3 = _mm512_log_ps(_mm512_mask_blend_ps(
22       mask1, ones, z2));
23     h_all = _mm512_fmadd_ps(z2, z3, h_all);
24     z0 = _mm512_load_si512(table.ctrls + i);
25     z3 = _mm512_mul_ps(
26       _mm512_cvtepi32_ps(z0), ii);
27     // Identify cells with 0's
28     mask2 = _mm512_cmp_epi32_mask(z0,
29       _mm512_setzero_si512(), _MM_CMPINT_NE);
30     // Replace 0's with 1's before log
31     z4 = _mm512_log_ps(_mm512_mask_blend_ps(
32       mask2, ones, z3));
33     h_all = _mm512_fmadd_ps(z3, z4, h_all);
34     z1 = _mm512_add_ps(z2, z3);
35     // Merge previous masks
36     mask3 = _kor_mask16(mask1, mask2);
37     // Replace 0's with 1's before log
38     z2 = _mm512_log_ps(_mm512_mask_blend_ps(
39       mask3, ones, z1));
40     h_x = _mm512_fmadd_ps(z1, z2, h_x);
41   }
42
43   return _mm512_reduce_add_ps(h_all) -
44     _mm512_reduce_add_ps(h_x) - h_y;
45 }
```

References

[1] G. Churchill, Epistasis, in: S. Maloy, K. Hughes (Eds.), Brenner's Encyclopedia of Genetics (Second Edition), second ed., Academic Press, San Diego, 2013, pp. 505–507, http://dx.doi.org/10.1016/B978-0-12-374984-0.00482-4.

[2] S. He, J.C. Reif, V. Korzun, R. Bothe, E. Ebmeyer, Y. Jiang, Genome-wide mapping and prediction suggests presence of local epistasis in a vast elite winter wheat populations adapted to Central Europe, Theor. Appl. Genet. 130 (4) (2017) 635–647.

[3] Y. Jiang, R.H. Schmidt, Y. Zhao, J.C. Reif, A quantitative genetic framework highlights the role of epistatic effects for grain-yield heterosis in bread wheat, Nature Genet. 49 (12) (2017) 1741–1746.

[4] P. Banerjee, V.A.O. Carmelo, H.N. Kadarmideen, Genome-Wide Epistatic Interaction Networks Affecting Feed Efficiency in Duroc and Landrace Pigs, Front. Genet. 11 (2020) 121.

[5] O. Ruiz-Larrañaga, P. Vázquez, M. Iriondo, C. Manzano, M. Aguirre, J.M. Garrido, R.A. Juste, A. Estonba, Evidence for gene-gene epistatic interactions between susceptibility genes for Mycobacterium avium subsp. paratuberculosis infection in cattle, Livest. Sci. 195 (2017) 63–66.

[6] J.J. Meijsen, A. Rammos, A. Campbell, C. Hayward, D.J. Porteous, I.J. Deary, R.E. Marioni, K.K. Nicodemus, Using tree-based methods for detection of gene–gene interactions in the presence of a polygenic signal: simulation study with application to educational attainment in the Generation Scotland Cohort Study, Bioinformatics (ISSN: 1367-4803) 35 (2) (2018) 181–188, http://dx.doi.org/10.1093/bioinformatics/bty462.

[7] A. Wollstein, S. Walsh, F. Liu, U. Chakravarthy, M. Rahu, J.H. Seland, G. Soubrane, L. Tomazzoli, F. Topouzis, J.R. Vingerling, et al., Novel quantitative pigmentation phenotyping enhances genetic association, epistasis, and prediction of human eye colour, Sci. Rep. 7 (1) (2017) 1–11.

[8] Y.-H. Kim, Y. Yoon, Y.-H. Kim, Towards a Better Basis Search through a Surrogate Model-based Epistasis Minimization for Pseudo-Boolean Optimization, Mathematics 8 (8) (2020) 1287.

[9] J. Shang, J. Zhang, Y. Sun, Y. Zhang, EpiMiner: A three-stage co-information based method for detecting and visualizing epistatic interactions, Digit. Signal Process. 24 (2014) 1–13.

[10] Y. Sun, J. Shang, J.-X. Liu, S. Li, C.-H. Zheng, epiACO - A method for identifying epistasis based on ant Colony optimization algorithm, BioData Min. 10 (1) (2017) 1–17.

[11] J. Wang, T. Joshi, B. Valliyodan, H. Shi, Y. Liang, H.T. Nguyen, J. Zhang, D. Xu, A Bayesian model for detection of high-order interactions among genetic variants in genome-wide association studies, Bmc Genom. 16 (1) (2015) 1–20.

[12] C. Ponte-Fernandez, J. Gonzalez-Dominguez, A. Carvajal-Rodriguez, M.J. Martin, Evaluation of Existing Methods for High-Order Epistasis Detection, IEEE/ACM Trans. Comput. Biol. Bioinform. (2020) http://dx.doi.org/10.1109/TCBB.2020.3030312.

[13] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N.L. Tang, W. Yu, BOOST: A Fast approach to detecting gene-gene interactions in genome-wide case-control studies, Am. J. Hum. Genet. 87 (3) (2010) 325–340, http://dx.doi.org/10.1016/j.ajhg.2010.07.021.

[14] R. Campos, D. Marques, S. Santander-Jiménez, L. Sousa, A. Ilic, Heterogeneous CPU+iGPU processing for efficient epistasis detection, 2020.

[15] H. Martínez, S. Barrachina, M. Castillo, E.S. Quintana-Orti, J. Rambla de Argila, X. Farré, A. Navarro, FaST-LMM for Two-Way Epistasis Tests on High-Performance Clusters, J. Comput. Biol. 25 (8) (2018) 862–870.

[16] C. Ponte-Fernández, J. González-Domínguez, M.J. Martín, Fast search of third-order epistatic interactions on CPU and GPU clusters, Int. J. High Perform. Comput. Appl. (2019) 20–29, http://dx.doi.org/10.1177/1094342019852128.

[17] J. González-Domínguez, B. Schmidt, GPU-Accelerated exhaustive search for third-order epistatic interactions in case–control studies, J. Comput. Sci. 8 (2015) 93–100.

[18] R. Nobre, A. Ilic, S. Santander-Jiménez, L. Sousa, Exploring the Binary Precision Capabilities of Tensor Cores for Epistasis Detection, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020, pp. 338–347, http://dx.doi.org/10.1109/IPDPS47924.2020.00043.

[19] L. Wienbrandt, J.C. Kässens, J. González-Domínguez, B. Schmidt, D. Ellinghaus, M. Schimmler, FPGA-based Acceleration of Detecting Statistical Epistasis in GWAS, Procedia Comput. Sci. 29 (2014) 220–230.

[20] G.R. Luecke, N.T. Weeks, B.M. Groth, M. Kraeva, L. Ma, L.M. Kramer, J.E. Koltes, J.M. Reecy, Fast Epistasis Detection in Large-Scale GWAS for Intel Xeon Phi Clusters, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 3, 2015, pp. 228–235, http://dx.doi.org/10.1109/Trustcom.2015.637.

[21] S. Gálvez, F. Agostini, J. Caselli, P. Hernandez, G. Dorado, BLVector: FAst BLAST-like algorithm for manycore CPU with vectorization, Front. Genet. 12 (2021).

[22] E. Rucci, C.G. Sanchez, G.B. Juan, A. De Giusti, M. Naiouf, M. Prieto-Matias, SWIMM 2.0: enhanced Smith–Waterman on Intel'S multicore and manycore architectures based on AVX-512 vector extensions, Int. J. Parallel Program. 47 (2) (2019) 296–316.

[23] Z. Yin, X. Xu, J. Zhang, Y. Wei, B. Schmidt, W. Liu, RabbitMash: Accelerating hash-based genome analysis on modern multi-core architectures, Bioinformatics (2020).

[24] X. Guo, Y. Meng, N. Yu, Y. Pan, Cloud computing for detecting high-order genome-wide epistatic interaction via dynamic clustering, BMC Bioinformatics 15 (1) (2014) 1–16.

[25] W. Muła, N. Kurz, D. Lemire, Faster population counts using AVX2 instructions, Comput. J. 61 (1) (2018) 111–120.

[26] W. Muła, GitHub repository "SIMD popcount", containing the vector algorithms published in [25], 2016. https://github.com/WojciechMula/sse-popcount (Accessed: 2021-02-24).

[27] I. Corporation, Second Generation Intel Xeon Scalable Processors Specification Update, 2020, https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/2nd-gen-xeon-scalable-spec-update.pdf (Accessed: 2020-11-07).

Christian Ponte-Fernández is a Ph.D. student in Information Technology Research at Universidade da Coruña, Spain. He received his B.Sc. degree in Computer Engineering and M.Sc. degree in Bioinformatics for Health Sciences from the Universidade da Coruña in 2016 and 2018, respectively. His research interests include High Performance Computing, and parallel and distributed algorithms applied to Bioinformatics.

Jorge González-Domínguez received his B.Sc., M.Sc., and Ph.D. degrees in Computer Science from the Universidade da Coruña, Spain, in 2008, 2010, and 2013, respectively. He is currently an assistant teacher in the Computer Architecture Group at the Universidade da Coruña, Spain. His main research interests are in the areas of high-performance computing for Bioinformatics and PGAS programming languages.

María J. Martín received her BS (1993), MS (1994), and Ph.D. (1999) degrees in Physics from the University of Santiago de Compostela, Spain. Since 1997, she has been on the faculty of the Department of Computer Engineering at the Universidade da Coruña. Her major research interests include parallel algorithms and applications and fault tolerance for parallel applications.