Ultra-Fast Detection of Higher-Order Epistatic Interactions on GPUs

Daniel Jünger[1], Christian Hundt[1], Jorge González-Domínguez[2(✉)], and Bertil Schmidt[1]

[1] Institut für Informatik, Johannes Gutenberg-Universität Mainz, Mainz, Germany
djuenger@students.uni-mainz.de, {hundt,bertil.schmidt}@uni-mainz.de
[2] Grupo de Arquitectura de Computadores, Universidade da Coruña, A Coruña, Spain
jgonzalezd@udc.es

Abstract. Detecting higher-order epistatic interactions in Genome-Wide Association Studies (GWAS) remains a challenging task in the fields of genetic epidemiology and computer science. A number of algorithms have recently been proposed for epistasis discovery. However, they suffer from a high computational cost since statistical measures have to be evaluated for each possible combination of markers. Hence, many algorithms use additional filtering stages discarding potentially non-interacting markers in order to reduce the overall number of combinations to be examined. Among others, Mutual Information Clustering (MIC) is a common pre-processing filter for grouping markers into partitions using K-Means clustering. Potentially interacting candidates for high-order epistasis are then examined exhaustively in a subsequent phase. However, analyzing real-world datasets of moderate size can still take several hours when performing analysis on a single CPU. In this work we propose a massively parallel computation scheme for the MIC algorithm targeting CUDA-enabled accelerators. Our implementation is able to perform epistasis discovery using more than 500,000 markers in just a couple of seconds in contrast to several hours when using the sequential MIC implementation. This runtime reduction by two orders-of-magnitude enables fast exploration of higher-order epistatic interactions even in large-scale GWAS datasets.

Keywords: Bioinformatics · GWAS · Epistasis · High performance computing · CUDA

1 Introduction

Discovering genotype-phenotype associations between genetic markers and certain diseases has become an increasing field of interest in recent years. Case-control studies, such as Genome Wide Association Studies (GWAS), search for genetic factors that influence common complex traits. Some of these studies have

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 421–432, 2017.
DOI: 10.1007/978-3-319-58943-5_34

explored single-locus associations between specific markers and a certain disease [4,5]. However, most complex diseases are suspected to have more sophisticated association patterns [1]. One cause of complex association patterns arises from the existence of epistasis; i.e. interactions among k markers ($k \geq 2$). A variety of algorithms has been proposed using different approaches for finding such epistatic interactions in GWAS. Exhaustive search approaches [6,7,10,11] examine every possible k-combination of markers. Hence, these approaches promise high accuracy but often lack scalability, since the number of possible combinations grows exponentially with the order of interaction k. Stochastic random sampling methods [14] usually need to specify many parameters that heavily influence their execution time. Machine learning algorithms [9,12] are often faster than exhaustive approaches, but may only find local extrema instead of globally optimal solutions.

Approaches for finding higher-order epistasis in GWAS use filter cascades such as SNPHarvester [13] or MIC [8]. These approaches utilize filters to prune unpromising markers that are unlikely to exhibit high interactions. Subsequently, the markers that have survived the filtration stage are examined exhaustively for k-locus interactions. The MIC algorithm uses K-Means clustering in combination with mutual information as distance measure for the filtering to determine sets of markers that are potentially interacting. Afterwards, the obtained candidates are examined exhaustively. Using a CPU-only implementation of MIC, it is possible to search for six-SNPs epistasis in the well-known Wellcome Trust Case-Control Consortium (WTCCC) dataset with over 500,000 markers in a couple of hours.

The main contributions of this paper are the design of fine-grained parallelization schemes for the sequential MIC algorithm targeting massively parallel architectures and their implementation on CUDA-enabled accelerators providing speedups of around two orders-of-magnitude in comparison to single-threaded CPU code. Consequently, we are able to reduce the runtime of MIC drastically from hours to seconds enabling researchers to perform exploratory analysis in an interactive manner.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the sequential MIC algorithm. Section 3 describes our parallelization scheme. Performance is evaluated in Sect. 4. Section 5 concludes the paper.

2 Background

Mutual Information Clustering (MIC) performs fast candidate selection for higher-order epistatic interactions in GWAS. It consists of two stages for detecting k-locus interactions. The first stage filters single-nucleotide polymorphisms (SNPs) that are unlikely to interact using a variant of K-Means clustering that determines a notion of similarity by the pairwise computation of mutual information between the individual markers. After the clustering step a user-defined number of m SNP candidates are selected from each cluster. These candidates are examined to find the causative SNPs of k-locus interactions.

Based on [8], mutual information I is used as similarity measure of association between genotypes and susceptibilities of diseases. Let $X = \{A_1, A_2, \ldots, A_n\}$ be

a partition of a set S, meaning that $S = A_1 \cup A_2 \cup \ldots \cup A_n$ and $A_i \cap A_j = \emptyset$ for all distinct pairs of i and j. The entropy $H(X)$ can be expressed as

$$H(X) = -\sum_{i=1}^{n} \frac{|A_i|}{|S|} \cdot \log \frac{|A_i|}{|S|} \tag{1}$$

where $|\cdot|$ denotes the number of elements in a set. Note that $\frac{|A_i|}{|S|}$ can be interpreted as the probability mass function of the partition X. An extension of this definition to an arbitrary number of partitions is straightforward. Let $X_j = \{A_1^{(j)}, A_2^{(j)}, \ldots, A_n^{(j)}\}$ for $j = 1, \ldots, k$ be k partitions of a set S. Then the joint entropy of k partitions $H(X_1, X_2, \ldots, X_k)$ is defined as

$$H(X_1, X_2, \ldots, X_k) = -\sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_k=1}^{n_k} P_{i_1\, i_2 \cdots i_k} \cdot \log P_{i_1\, i_2 \cdots i_k}$$

$$\text{where} \qquad P_{i_1\, i_2 \cdots i_k} = \frac{|A_{i_1}^{(1)} \cap A_{i_2}^{(2)} \cap \ldots \cap A_{i_k}^{(k)}|}{|S|} \quad . \tag{2}$$

The mutual information between the joined partition of $X_1, X_2, \ldots, X_k$ and a partition Y can be expressed as:

$$\begin{aligned} I(X_1, X_2, \ldots, X_k; Y) = H(Y) + H(X_1, X_2, \ldots, X_k) \\ - H(X_1, X_2, \ldots, X_k, Y) \end{aligned} \tag{3}$$

Let $X_1, X_2, \ldots, X_k$ be partitions for the set of samples induced by the genotypes of $\mathrm{SNP}_1, \mathrm{SNP}_2, \ldots, \mathrm{SNP}_k$, respectively, and Y be the partition by disease state (case or control) then $I(X_1, X_2, \ldots, X_k; Y)$ represents the degree of associations between genotypes of $\mathrm{SNP}_1, \mathrm{SNP}_2, \ldots, \mathrm{SNP}_k$ and the disease state. The objective is to find the set of k SNPs that maximizes the value of $I(X_1, X_2, \ldots, X_k; Y)$. Examining every possible k-combination of n SNPs is considered computational intractable for more than half a million SNPs in GWAS for $k \geq 3$ [6,7]. In order to reduce the number of SNPs to be considered in the exhaustive step, MIC uses K-Means which scales linearly in the number of processed markers. The clustering procedure is a modification of Lloyd's algorithm:

1. *Assignment step.* The pair-wise distance $\mathrm{dist}(X_i, X_j)$ between two SNPs X_i and X_j is defined as the mutual information $I(X_i, X_j; Y)$. This implies that SNPs that are strongly interacting tend to be placed into different clusters.
2. *Update step.* The process of selecting the centroid of each cluster works as follows. Each SNP generates a contingency table consisting of genotype frequencies among samples. The average contingency table $T_{\mathrm{avg}}^{(j)}$ of a cluster j is defined as follows: each entry of $T_{\mathrm{avg}}^{(j)}$ is the average of the corresponding entries of all contingency tables generated by the SNPs belonging to the cluster j. A centroid c_j of a cluster j is defined as the nearest neighbour of $T_{\mathrm{avg}}^{(j)}$ with respect to the sum of squared errors

$$c_j = \underset{q}{\operatorname{argmin}} \, \|T_q - T_{\mathrm{avg}}^{(j)}\|^2. \tag{4}$$

After the clustering step, m candidates are selected from each cluster. A candidate in a cluster is a SNP that is far apart (in terms of pairwise mutual information) from SNPs in other clusters. MIC makes use of this similarity measure to define a score value for SNPs. Let $x^{(i)}$ be a SNP in the i-th cluster with c_i as the corresponding centroid then the score value is determined by:

$$\begin{aligned}\text{score}(x^{(i)}) &= \sum_{j \neq i} \text{dist}(x^{(i)}, c_j) \\ &= I(x^{(i)}, c_1 \ \ ; Y) + \ldots + I(x^{(i)}, c_{i-1}; Y) \\ &+ I(x^{(i)}, c_{i+1}; Y) + \ldots + I(x^{(i)}, c_k; Y). \end{aligned} \tag{5}$$

From each cluster MIC selects the top m SNPs in terms of their scores as candidates for further processing. Thus, a total of $k \cdot m$ candidates are chosen. Among these candidates, MIC exhaustively searches the k-tuple with the highest mutual information value $I(X_1, X_2, \ldots, X_k; Y)$. This implies that it only has to probe $\binom{m \cdot k}{k}$ combinations instead of $\binom{n}{k}$, where $m \cdot k \ll n$.

3 CUDA Implementation

In this section, we discuss the details of our parallel implementation of the MIC algorithm using CUDA. Besides native CUDA, we also utilize the CUDA Unbound (CUB) library [3] which provides a set of highly optimized parallel primitives. We subdivide the MIC algorithm into the following four distinct phases.

3.1 Data Preparation

Our implementation stores the genotype information in form of a C++ standard library vector containing SNP elements. A SNP is represented by a struct containing the genotype information for both cases and controls. SNPs are expressed in three different genotypes for both cases and controls. Hence, the SNP-struct has six sub-elements. Each of these sub-elements is a bit-array where the bit at index i encodes whether the ith individual (case or control) has the particular genotype. For simple enumeration we will label genotype AA as 0, AB as 1, and BB as 2. Hence, we can refer to the genotype arrays of the SNP-struct as case0, case1, case2, ctrl0, ctrl1, and ctrl2. For later use in the clustering step we pre-compute the genotype frequencies of each SNP for cases and controls respectively, by determining the population count of each bit array. This step takes linear time. We refer to the structure of combinations of the six genotype frequencies as the *contingency table*.

In order to use the genotype information on the GPU in an efficient manner, we use six global bit-arrays on the CUDA device, each of which combines one genotype case/control bit array from all SNPs one after another. Subsequently, we transpose each bit-array, assuring coalesced access between CUDA threads in

a warp if threads are assigned to SNPs within the SNP set consecutively. Transposition is achieved using one CUDA-stream per array using a shared memory-based out-of-place transposition algorithm. We compute the genotype frequency of each SNP using the vectorization capabilities of the GPU along with coalesced data access patterns.

3.2 Clustering

The modified K-Means algorithm can be split into three subroutines that are parallelized separately using dedicated CUDA kernels.

First, the cluster assignment step compares each SNP with the set of centroids c_j and subsequently assigns the nearest neighbour. Since this can be determined for each of the SNPs independently, we map individual SNPs to CUDA-threads exploiting the optimized data alignment discussed in the previous subsection. As a result, the cluster indices of each SNP are stored in an array residing in the global memory of the GPU.

Second, the mean contingency table of each cluster is computed by pointwise addition of all contingency tables of SNPs that are assigned to that cluster and subsequent division by the number of SNPs in the cluster. The applied reduction algorithm utilizes different memory spaces of the GPU. On the lowest level each warp (consisting of 32 threads) computes its partial result using warp intrinsics and stores the result in the shared memory of its block. Subsequently, each block uses a tree-based reduction to accumulate partial sums and stores the final result in global memory using atomic operations. We then divide the per-cluster accumulated contingency table by the number of SNPs in each cluster in parallel using the device-wide `cub::DeviceHistogram` primitive from the CUB library on the cluster array in order to determine the cluster sizes.

The third subroutine determines the updated centroids for the next iteration of Lloyd's algorithm by computing the nearest neighbour SNP of each centroid in terms of sum of squared errors to the mean contingency table of the corresponding cluster. In order to update the centroids in parallel, we first compute the distance of each SNP to its corresponding cluster mean using one thread per SNP. Subsequently, the obtained distance values are stored as 32-bit unsigned integer into the lower half of a 64-bit unsigned integer and consecutively write the 8-bit cluster identifier of a SNP into the upper half. This step is visualized in Fig. 1(a). We can now define a lexicographical ordering over these elements with the cluster identifier as major order and the distance value as minor order. Using this relation, we sort this array using a device-wide call to `cub::DeviceRadixSort`. A schematic overview of this step is illustrated in Fig. 1(b). Note that CUB provides the ability to run radix-sort only on a sub-set of bits of an integer. Hence, we just consider the first 40 bits of an element for the sorting step. The SNP of cluster c_j with minimal distance to the mean contingency table is placed at index $\sum_{i=1}^{j-1} |c_i|$. We then use a `cub::DeviceExclusiveSum` primitive on the clustering histogram to determine the starting indices of each cluster. Finally, we select the first SNP of each cluster from the sorted array as the new centroid.

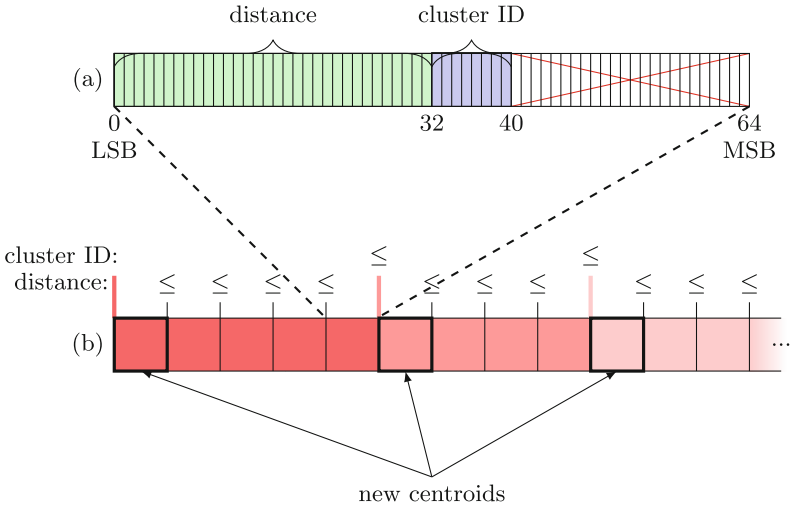

Fig. 1. Selection of centroids using lexicographical ordering. (a) Shows a 64-bit unsigned integer which represents a SNP. The distance from the SNP to its centroid is stored as a 32-bit unsigned integer in the lower half of the 64-bit datatype. The 8-bit long identifiers of the corresponding cluster are stored consecutively. Overall the struct holds 40 bits of information. (b) Shows the result of `cub::DeviceRadixSort` on an array of the datatype depicted in (a). The new centroid elements are the first elements of each cluster section (denoted by different color shading).

3.3 Candidate Selection

The score computation of each SNP can be performed independently by utilizing CUDA-threads. The candidates of each cluster are those m SNPs with the highest scores. For this purpose, we use a slight modification of the major-minor radix-sort approach as shown in Fig. 1. Different from our initial definition, we now pack the score value of type float into the lower half of a 64-bit unsigned integer together with the 8-bit cluster ID stored consecutively. Since we want to sort the elements of this array ascending by the cluster ID (major ordering) but descending by the score values (minor ordering), we negate the score values before sorting the array. The selection step is analogous to Fig. 1(b): the first m SNPs of each cluster are selected in the sorted array rather than just one. As a result of this phase we have selected $k \cdot m$ SNPs that are exhaustively examined in the final phase.

3.4 Exhaustive Search

Algorithm 1 represents the implementation of Eq. 3. As a subtask of this calculation, we probe each of the 3^k possible genotype combinations of the given SNP combination in order to determine the joint entropies $H(X_1, X_2, \ldots, X_k)$ and $H(X_1, X_2, \ldots, X_k, Y)$ as given in Eq. 2. For one of these genotype combinations

$i_{geno} \in \{0, \ldots, 3^k - 1\}$ the genotype $g_{i_k} \in \{0, 1, 2\}$ to choose for one SNP snp_{i_k} with $i_k \in \{0, \ldots, k-1\}$ of this SNP combination can be calculated by:

$$g_{i_k} = \lfloor \frac{i_{geno}}{3^{i_k}} \rfloor \mod 3 \tag{6}$$

Using this extension we can now implement the k-locus mutual information for one SNP combination as follows:

Algorithm 1. Mutual Information of k loci

```
1: procedure KMI
2:     pCase← 0.0
3:     pCtrl← 0.0
4:     H_xy ← 0.0
5:     H_x ← 0.0
6:     H_y ← H(Y)                                   ▷ computation according to Eq. 1
7:
8:     for i_geno ∈ [0, 3^k) do                     ▷ 3^k combinations of genotypes
9:         f_case←POPC(snp_0.cases[g_0]∩snp_1.cases[g_1]∩ ... ∩ snp_{k-1}.cases[g_{k-1}])
10:        f_ctrl←POPC(snp_0.ctrls[g_0]∩snp_1.ctrls[g_1]∩ ... ∩ snp_{k-1}.ctrls[g_{k-1}])
11:
12:        pCase← f_case/(|cases| + |ctrls|)
13:        pCtrl← f_ctrl/(|cases| + |ctrls|)
14:
15:        H_x− = (pCase + pCtrl) · log (pCase + pCtrl)   ▷ computation according to
   Eq. 2
16:        H_xy− = pCase · log pCase + pCtrl · log pCtrl
17:    end for
18:    return H_y + H_x − H_xy                      ▷ computation according to Eq. 3
19: end procedure
```

Note that the computation of the joint frequencies is performed efficiently by using bitwise AND-operations followed by a CUDA-intrinsic population count on the SNP bit-sets (see Lines 9 and 10 in Algorithm 1). The parallelization-scheme for this task assigns each SNP combination to one CUDA-thread.

The task of computing the k-locus mutual information for each k-combination is computationally demanding. We reduce the computational load per CUDA core by pre-computing the SNP combinations of the given candidates as follows.

First, we need to find a mapping that associates each combination index $i_{comb} \in \{1, \ldots, \binom{k \cdot m}{k}\}$ with a distinct k-combination from the set of $k \cdot m$ candidates. This can be implemented by decomposing binomial coefficients using their recursive definition:

$$\begin{aligned} \text{(I)}: \quad & \binom{n}{n} = \binom{n}{0}; \\ \text{(II)}: \quad & \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}; \end{aligned} \tag{7}$$

If we substitute n by $(km-1)$ and $k\ b < (k-1)$ we can rewrite (II) as

$$\binom{k \cdot m}{k} = \binom{km-1}{k-1} + \binom{km-1}{k} \tag{8}$$

Using this representation we can apply a recursive binary tree decomposition. Each level of the tree represents one element of the set of km elements. Additionally, each distinct path through the tree represents a distinct SNP combination. Algorithm 2 computes one path given the index i_{comb} of the combination to be formed and returns the corresponding k-combination. We will execute this algorithm on $\binom{k \cdot m}{k}$ CUDA-threads, each one processing a single combination.

Algorithm 2. Computation of k-combination

```
 1: procedure getSNPCombination(i_comb)
 2:     combination[]
 3:     index← i_comb
 4:     local_n← k · m
 5:     local_k← k
 6:     j ← 0
 7:
 8:     for i ∈ [0, km) do
 9:         lower← (local_n−1 choose local_k)
10:
11:         if index ≥ lower then
12:             local_k -= 1
13:             combination[j]← i
14:             j++
15:             index -= lower
16:         end if
17:         local_n -= 1
18:     end for
19:     return combination[]
20: end procedure
```

Algorithm 2 calls the binomial coefficient function $k \cdot m$ times per thread in Line 9. To further reduce the workload of each CUDA-core, we pre-compute the values of the binomial coefficients and cache them in a look-up table residing in global memory. Finally, we determine the highest epistatic interaction candidate using a device-wide key-value sort primitive form the CUB library.

4 Experimental Evaluation

In order to measure the time benefits of our CUDA-parallelized version of MIC compared to the single- and multi-core CPU version, we use a real-world dataset from WTCCC. The dataset consists of the genotype information of roughly

500,000 SNPs that were gathered from 3,000 controls drawn from the British population and 2,000 cases which are all affected by *inflammatory bowel disease*. The system configuration used for benchmarking is listed in Table 1.

Table 1. Benchmark system.

Host system	CPU	Intel Core i7-3970X, 64-bit, HT
	CPU cores	6 cores @ 3.50 GHz (max. 4.0 GHz)
	RAM	32 GB DDR3
	OS	Ubuntu 14.04.4 LTS, 64-bit
CUDA device	Device	NVIDIA GeForce GTX Titan X
	GPU	NVIDIA GM 200
	GPU cores	3072 SPs @ 1GHz
	DRAM	12 GB GDDR5
	CC	5.2
Compilers	Host	g++ v4.8.4
	Device	nvcc v7.5.17
Compiler flags	g++	-O3 -std=c++11 -fopenmp
	nvcc	-O3 –expt-relaxed-constexpr -use_fast_math
		-std=c++11 -rdc true
		-gencode=arch=compute_52,code=sm_52

In this work we focus on testing the performance improvement of our GPU-based parallel implementation as the accuracy is the same as the original MIC which has already been assessed in [8]. The MIC algorithm can be divided into two major phases. The first phase represents the K-Means clustering step. This step takes $\mathcal{O}(lkn)$ time, where l denotes the number of samples (cases/controls), k the number of clusters, and n the number of SNPs to be examined. The second phase performs exhaustive search and examines $\binom{k \cdot m}{k}$ k-combinations of SNPs for epistasis. The computation of the mutual information of each k-SNP combination and disease state takes linear time i.e. $\mathcal{O}(l)$. Thus, this phase takes $\mathcal{O}(l(km)^k)$ time. Since k occurs in the asymptotical runtime of both phases, we choose k as the varying parameter for our benchmark. The value of n is given by the WTCCC dataset and therefore fixed. We also set $m = 5$ throughout the experiments.

Table 2 shows the benchmark results for varying values of k (from one to six). As the original MIC implementation [8] is not publicly available, we developed a CPU-based C implementation and parallelized it using Open Multi-Processing (OpenMP or OMP)[2] for comparison purposes. We use the average of 50 executions for the GPU implementation and 30 executions for the CPU implementations. However, sequential execution for the highest k takes more than two hours and is not very stable at runtime. Hence, we were only able to measure two executions for this configuration with the sequential implementation.

Table 2. Average runtimes in seconds and speedups of the CUDA implementation on a GTX Titan X GPU over a single- and multi-core CPU-based version for $m = 5$.

	k	2	3	4	5	6
Runtime	t_{seq}	11.39	26.97	183.75	386.58	7865.17
	t_{omp}	4.25	5.11	6.67	49.44	1926.42
	t_{cuda}	0.69	0.74	0.83	1.36	16.41
Speedup	t_{seq}/t_{omp}	2.68	5.28	27.55	7.82	4.08
	t_{seq}/t_{cuda}	37.85	36.41	221.66	285.00	479.30
	t_{omp}/t_{cuda}	6.16	6.91	8.04	36.35	117.39

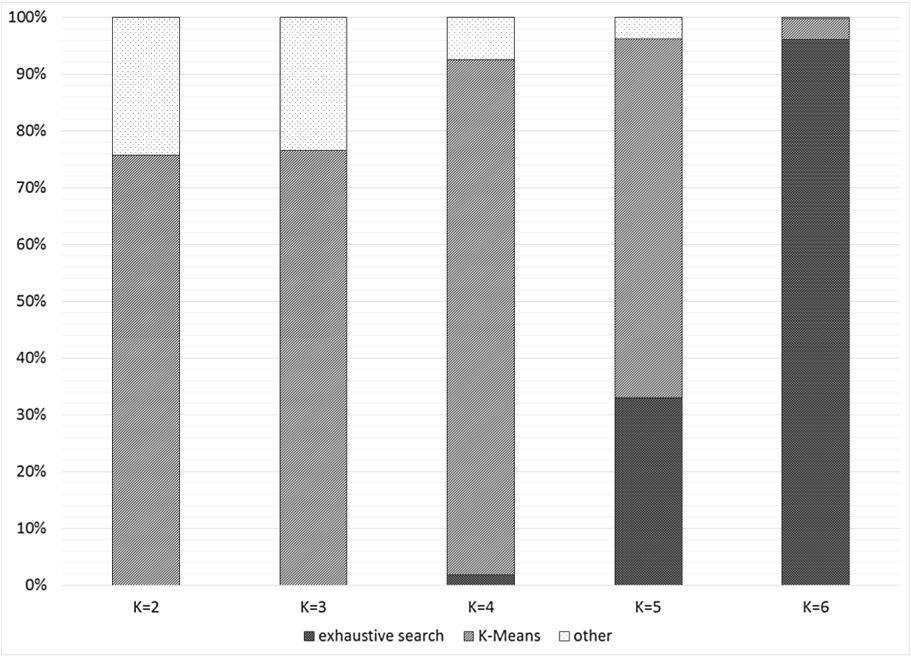

Fig. 2. Execution time proportions of K-Means and exhaustive step.

The benchmark results show that the speedups grow when k is increased. This is due to the fact that the number of combinations in the exhaustive search step grows exponential, as the asymptotic runtime is given by $\mathcal{O}(l(km)^k)$. Figure 2 illustrates the proportions between the K-Means step and the exhaustive step to the execution time when k is increased. We observe that K-Means holds the largest share for $k \leq 5$, whereas the exhaustive step, by far, holds the biggest share for $k > 5$. Our CUDA-based approach obtains more benefit for experiments where the exhaustive search has a significant impact on the total runtime.

5 Conclusion

We have developed an efficient parallel implementation of the MIC algorithm for finding higher-order epistasis in GWAS using CUDA-enabled accelerator cards. Concretely, we have proposed a parallel GPU-only implementation of a modified K-Means clustering algorithm. In addition to the clustering step, we also make extensive use of the GPU for the remaining parts, leaving the host CPU only for organizational purposes during execution.

Using our implementation it is possible to examine moderately-sized GWAS datasets in just a few seconds on a modern consumer-grade workstation. Our benchmark results indicate speedups of about two orders-of-magnitude compared to the sequential solution. The benefits of our parallel implementation are more significant when increasing the order of the interactions, i.e. when the exhaustive phase has more impact on the total execution time.

As part of our future work, we are planning to further improve the CUDA-implementation of the exhaustive search step. For now, this computation is done by a so-called *heavy kernel*, where each thread has to compute a rather big portion of the overall task. The CUDA architecture, however, is designed and optimized for *lightweight threads*. Hence, we have to develop a parallelization-scheme that implements the concept of lightweight threads by further splitting each computation into independent subtasks. A further possible direction of future research is the design and comparison of novel lightweight candidate selection algorithms on CUDA-enabled accelerators in order to robustly prune non-interacting markers at even higher speed.

Acknowledgments. This study makes use of data generated by the Wellcome Trust Case-Control Consortium. A full list of the investigators who contributed to the generation of the data is available from www.wtccc.org.uk. Funding for the project was provided by the Wellcome Trust under award 076113 and 085475.

References

1. Cordell, H.J.: Detecting gene-gene interactions that underlie human diseases. Nat. Rev. Genet. **10**(6), 392–404 (2009)
2. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
3. Duane Merrill, N.C.: Cub documentation (2016). https://nvlabs.github.io/cub/
4. Easton, D.F., Pooley, K.A., et al.: Genome-wide association study identifies novel breast cancer susceptibility loci. Nature **447**(7148), 1087–1093 (2007)
5. Frayling, T.M., Timpson, N.J., et al.: A common variant in the FTO gene is associated with body mass index and predisposes to childhood and adult obesity. Science **316**(5826), 889–894 (2007)
6. González-Domínguez, J., Schmidt, B.: GPU-accelerated exhaustive search for third-order epistatic interactions in case-control studies. J. Comput. Sci. **8**, 93–100 (2015)
7. Kässens, J.C., Wienbrandt, L., González-Domínguez, J., Schmidt, B., Schimmler, M.: High-speed exhaustive 3-locus interaction epistasis analysis on FPGAs. J. Comput. Sci. **9**, 131–136 (2015)

8. Leem, S., Jeong, H.H., et al.: Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure. Comput. Biol. Chem. **50**, 19–28 (2014)
9. Meng, Y.A., Yu, Y., et al.: Performance of random forest when SNPS are in linkage disequilibrium. BMC Bioinf. **10**(1), 1 (2009)
10. Nelson, M., Kardia, S., et al.: A combinatorial partitioning method to identify multilocus genotypic partitions that predict quantitative trait variation. Genome Res. **11**(3), 458–470 (2001)
11. Wan, X., Yang, C., et al.: Boost: a fast approach to detecting gene-gene interactions in genome-wide case-control studies. Am. J. Hum. Genet. **87**(3), 325–340 (2010)
12. Wan, X., Yang, C., et al.: Predictive rule inference for epistatic interaction detection in genome-wide association studies. Bioinformatics **26**(1), 30–37 (2010)
13. Yang, C., He, Z., et al.: SNPHarvester: a filtering-based approach for detecting epistatic interactions in genome-wide association studies. Bioinformatics **25**(4), 504–511 (2009)
14. Zhang, Y., Liu, J.S.: Bayesian inference of epistatic interactions in case-control studies. Nature Genet. **39**(9), 1167–1173 (2007)