METHODOLOGY

Efficient Calculation of the Genomic Relationship Matrix

Martin Schlather*

Correspondence:
schlather@math.uni-mannheim.de
Universität Mannheim, Institut für Mathematik, C8, 5, 68131 Mannheim, Germany
Full list of author information is available at the end of the article
*Equal contributor

Abstract

Background: The calculation of a genomic relationship matrix needs a large number of arithmetic operations. Therefore, fast implementations are of interest. The currently fastest implementations use AVX floating-point arithmetics.

Results: Our fastest algorithm is more accurate and $25\times$ faster than a AVX double precision floating-point implementation.

Conclusions: The spectrum of presented methods suggests that further improvement might be possible and that bit manipulation in combination with hash tables might be of relevance also for other calculation problems.

Keywords: crossproduct; covariance matrix; genetic relationship; genomic relatedness matrix; hash table

Background

The genomic relationship matrix (GRM) is the covariance matrix calculated from the SNP information of the individuals, i.e., from the minor allele counts [1]. It is an important ingredient in mixed models and generalized mixed models for analyses and predictions in genetics [2].

Let n be the number of individuals and s the number of SNPs per individual. Then, the calculation of the GRM needs of order sn^2 arithmetic operations. Most software packages use floating-point arithmetics, for instance the R packages `AGHmatrix` [3], `qgg` [4], `rrBLUP` [5], `snpReady` [6], and `GENESIS` [7]. The software `GCTA` [8] treats missings explicitly. The package `SNPRelate` [9] also uses floating-point arithmetics for the covariance matrix, but uses bit manipulation algorithms for other calculations (identity-by-descent estimates). PLINK [10] profits from bit manipulations for calculating the uncentred covariance matrix.

In this paper we present a couple of ideas, how the GRM can be calculated efficiently from a SNP matrix. We assume that no values are missing. The emphasis will be on algorithms that allow for a vectorized implementation (SIMD) and that take into account that the entries of the SNP matrix are most efficiently coded by 2 bits, namely for the values 0, 1 and 2, as they are present in diploid organism under the common assumption of biallelic markers.

Results

The standard mathematical formula for the GRM requires floating point arithmetics. An algebraic reformulation shows that the cost intensive part involves only integers. Since the most elementary numbers need only a minimum of 2 bits, a diversity of approaches for the integer arithmetics is thinkable. The investigated methods are in brief:

- `Multiply` : uses 16-bit arithmetics
- `Hamming2` : uses pop counts (the number of bits that are 1)
- `ThreeBit` : uses a 3-bit representation and a single large hash table
- `TwoBit` : uses the 2-bit representation and two large hash tables
- `Shuffle` : similar to `TwoBit`, but with two tiny hash tables
- `Packed` : uses 4-bit arithmetics

They are all available through `crossprodx` in the package `miraculix` [11]. Tables 1 and 2 show that `Shuffle` is the fastest method, which is, in the SSSE3 implementation, 35$\times$ faster than `crossprod` of `R` [12].

Table 1 Accelaration of the calculation of $M^\top M$ by `crossprodx`.

	Shuffle	Packed	Hamming2	Multiply	ThreeBit	TwoBit
acceleration	35×	28×	24×	17×	17×	15×
SIMD	SSSE3	SSE2	SSE2	SSE2	SSE2	none

The reference point is `crossprod` in `R`.

Table 2 Accelaration of the calculation of $M^\top M$ by AVX2 implementations in `crossprodx`.

	Shuffle	Packed	Multiply	AVX (double)	AVX2 (32-bit integer)
acceleration	48 ×	36×	29×	1.8×	4×

The reference point is `crossprod` in `R`.

The command `relationshipMatrix` in `miraculix` [11] for calculating the GRM is only negligibly slower than `crossprodx`. The AVX2 variant of `Shuffle` is even 48$\times$

faster than crossprod [12] and $48/1.8 \approx 25\times$ faster than a standard AVX double precision implementation for calculating the crossproduct of an arbitrary matrix, cf. Table 2. Furthermore, our algorithms have not even any cumulative rounding error.

Tables 1 and 2 also show that the AVX2 performance is hard to predict from the SSE performance. AVX2 variants for TwoBit and ThreeBit are not given since full vectorization is not possible. Hamming2 has not been pursued because of its memory demand.

For the benchmarks, we used an $s \times n$ SNP matrix with $n = 1000$ individuals and $s = 5 \cdot 10^5$ SNPS. The calculations were performed on an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz with R version 3.6.0 on Xubuntu. Although the code in miraculix is parallelized, we used only a single core for the benchmarks. Nonetheless, the AVX2 variant of Shuffle takes not more than 7 seconds.

The code for the benchmarks is available from the man page of crossprodx in miraculix.

Discussion

First, with respect to the memory needs of the SNP matrix, algorithms that use the 2-bit representation of a SNP value should be preferred. Among them, we have a sequence of distinct algorithms that differ in their speed-up and their SIMD requirements: TwoBit ($15\times$; SIMD not used), Packed ($28\times$; SSE2); Shuffle ($35\times$; SSSE3).

Second, the use of perfect hash tables to cut calculations short might be of general importance.

Third, since loading from non-aligned memory allocation is reported to be slower [13], the package miraculix was designed to avoid non-aligned loadings. Tests on the implemented package however show that the running time by non-aligned loadings is not reduced for SSE implementations. The speed is reduced by 5 to 10 % in AVX2 implementations. As the compressed SNP matrix is made available to the user as an R object and as the memory allocation by R is only 32-bit aligned, additional memory is allocated and the SNP matrix is aligned to 128 or 256 bits. Furthermore, additional zeros are appended so that the virtual number of SNPs is a multiple of the number of the SNPs that can be treated in a single step. The storing formats of `TwoBit`, `Multiply`, `Packed`, and `Shuffle`, including their AVX2 variants, were made compatible, i.e., the allocations are all based on a 256-bit alignment. A check and a reallocation are implemented for the case that memory is moved. This might happen when the garbage collector `gc` is called by `R`, for instance.

Conclusion

The combination of algebraic reformulation, bit manipulations and hash tables can reduce largely the computing time on SNP data. In the case of calculating the GRM, the computing time could be reduced by factor 25 in comparison to a straightforward AVX double precision implementation. As a spectrum of implementations exist, there is a chance of further improvement and of further applications of the underlying ideas.

Methods

Let M be an $s \times n$ SNP matrix of n individuals and s SNPs. We need to consider only the fast calculation of the crossproduct $M^\top M$, since the GRM A can be calculated from $M^\top M$ at low costs. This can be seen as follows.

Let $\mathbf{1}_k$ be the vector of length k whose components are all equal to 1. The centred and normalized GRM A is calculated as

$$A = (M - P)^\top (M - P)/\sigma^2$$

where

$$P = p\mathbf{1}_n^\top \quad \text{with} \quad p = \frac{1}{n} M \mathbf{1}_n.$$

and

$$\sigma^2 = \sum_{i=1}^{s} p_i(1 - p_i/2) \quad \text{with} \quad p = (p_1, \ldots, p_s).$$

Note that replacing the value p_i by the allele frequency $\tilde{p}_i = p_i/2$, we have the usual formula for σ^2,

$$\sigma^2 = 2\sum_{i=1}^{s} \tilde{p}_i(1 - \tilde{p}_i).$$

Let $B = M^\top M \mathbf{1}_n$. Then

$$n^2\sigma^2 A = n^2 M^\top M - n\mathbf{1}_n B^\top - nB\mathbf{1}_n^\top + \mathbf{1}_n \mathbf{1}_n^\top B \mathbf{1}_n^\top.$$

Hence, the integer-valued matrix $n^2\sigma^2 A$ can easily be calculated from the matrix $M^\top M$ without any numerical error and at low computational costs of order n^2. Now,

$$2n^2\sigma^2 = 2n^2 \sum_{i=1}^{s} p_i - n^2 \sum_{i=1}^{s} p_i^2 = 2n\mathbf{1}_s^\top M \mathbf{1}_n - \mathbf{1}_n^\top B.$$

Again, $2n^2\sigma^2$ can easily be calculated from M and $M^\top M$ without any numerical error. The computational costs are of order $n(n+s)$, hence still some magnitudes smaller than the costs of calculating the crossproduct $M^\top M$.

Algorithms for scalar products

Instead of considering the crossproduct $M^\top M$, it suffices to consider the scalar product of two vectors $a = (a_1, \ldots, a_s)$ and $b = (b_1, \ldots, b_s)$ whose components a_i and b_i have the values 0, 1 or 2. For simplicity and clarity, we will primarily refer to SSE commands in the following, and not to AVX.

Simple Multiplication

An immediate way of calculating the scalar product from a compressed 2-bit representation is to extract the first two bits of each of the two vectors a and b and to continue with integer arithmetics. Then the next two bits are extracted using shifting, and so on. Clearly, this procedure can be vectorized. Of particular advantage here is the SSE2 command `_mm_madd_epi16`, which multiplies and adds two consecutive 16-bit integers so that only 7 shifts are necessary for a vector of 64 SNP values, i.e., for 128 bits. We call this method `Multiply`.

Hamming Distance

The algorithm used in `PLINK` [14, 10] is based on the idea that a value is represented by the number of bits that equal 1 in a 4-bit representation. The values of the vectors a and b must be coded asymmetrically by two mappings f_a and f_b, say, as a coding by a single mapping is not possible. Then, the bitwise &-operator is applied before the number of 1's is counted. Table 3 gives a possible realisation.

Table 3 Table of values for the Hamming distance method.

$f_a(\cdot)$ & $f_b(\cdot)$	$f_b(0) = 0000_b$	$f_b(1) = 0011_b$	$f_b(2) = 1111_b$
$f_a(0) = 0000_b$	0000_b	0000_b	0000_b
$f_a(1) = 0110_b$	0000_b	0010_b	0110_b
$f_a(2) = 1111_b$	0000_b	0011_b	1111_b

The number of bits that equal 1 can be calculated by SSE2 commands based on work by [15, 16]. We call the method `Hamming2`. (An SSSE3 implementation in `miraculix` [11] is called `Hamming3`.) The method can be turned into a particularly fast implementation when novel AVX512 commands are used for the pop counts, e.g. `_mm512_popcnt_epi64`. See also `SNPRelate` [9] for pop count implementations. Still, the storing costs of the SNP matrix M remain high, namely $2 \times 4 = 8$ bits per SNP in a standard implementation.

Perfect Hash Table

Let us consider the product of the first elements a_1 and b_1 of the two SNP vectors a and b. Let us code the SNP values 0, 1 and 2 by 3 bits, e.g. as 000_b, 011_b and 110_b, respectively, and denote this mapping by f. Then, a perfect hash table for $f(a_1)$ & $f(b_1)$ returns 0, 1, 2, 4 for 000_b, 011_b, 010_b, and 110_b, respectively, cf. Table 4.

Table 4 Table of values for the `ThreeBit` method.

$f(\cdot)$ & $f(\cdot)$	$f(0) = 000_b$	$f(1) = 011_b$	$f(2) = 110_b$
$f(0) = 000_b$	000_b	000_b	000_b
$f(1) = 011_b$	000_b	011_b	010_b
$f(2) = 110_b$	000_b	010_b	110_b

Subvectors of length k of a and b can be treated in the same way and the perfect hash table returns then the scalar product of the subvectors. The hash table will be indexed by $3k$-bit numbers, i.e. by values between 0 and $2^{3k} - 1$. Since k should be as large as possible at a smallish size of the hash table, and $3k$ bits should fit nicely into 1, 2 or 4 bytes, the only reasonable choice for k is $k = 5$, so that 15 bits in a 16-bit representation of a vector with $k = 5$ components are used. The precise size of the hash table is then $110\,110\,110\,110\,110_b + 1 = 28087$ bytes. We call this method `ThreeBit`.

Two Perfect Hash Tables

Since we did not find a simple way to use a single hash table based on a 2-bit representation of the SNP values, we consider here two hash tables and the two bitwise operators & and | . The first hash table should return 1 and 4 for 01_b and 10_b, respectively, while the second hash table returns 2 for 11_b. All other values in the hash tables are 0, cf. Table 5.

Table 5 **Tables of values for the `TwoBit` method.**

$\&$	00_b	01_b	10_b
00_b	00_b	00_b	00_b
01_b	00_b	01_b	00_b
10_b	00_b	00_b	10_b

$\vert$	00_b	01_b	10_b
00_b	00_b	01_b	10_b
01_b	01_b	01_b	11_b
10_b	10_b	11_b	10_b

Then, the sum of the two table values yields the product $a_1 b_1$. Scalar products of subvectors of length k can also be treated by two hash tables. Since the size of both hash tables is of order 2^{2k}, one possible choice is $k = 8$, so that the size of the second hash table is 65536 bytes. We call this method `TwoBit`.

The disadvantage of `TwoBit` (and `ThreeBit`) is that the look-up in the hash table prohibits a full vectorization. A much better choice is therefore $k = 2$: the SSSE3 command `_mm_shuffle_epi8` looks 16 values up at once in a hash table of size 16. We call this variant `Shuffle`.

Packed arithmetics

A last idea is to emulate a multiplication by bitwise operations and partial sums. Let $\gg$ denote the bitwise shift operator and let

$$\begin{aligned}
c_i &= a_i \,\&\, b_i, \\
d_i &= (c_i \gg 1) \,\&\, 01_b, \\
e_i &= (a_i \gg 1) \,\&\, b_i \,\&\, 01_b, \text{ and} \\
f_i &= a_i \,\&\, (b_i \gg 1) \,\&\, 01_b,
\end{aligned}$$

so that $c_i + 2d_i = a_i b_i$ if $a_i = b_i$ and 0 else. Furthermore $2(e_i + f_i) = a_i b_i$ if $a_i b_i = 2$ and 0 else. In total, we have $c_i + 2d_i + 2(e_i + f_i) = a_i b_i$. Let $g_i = d_i + e_i + f_i$. Since $g_i = d_i \mid e_i \mid f_i$ for the bitwise operator $|$, only the values of g_i and c_i need to be summed up. An immediate extraction of the values of $g = (g_1, \ldots, g_s)$ and $c = (c_1, \ldots, c_s)$ by shifting as in the `Multiply` algorithm would be rather expensive. Instead, a 4-bit arithmetic can be introduced in an intermediate step for the four vectors

$$\begin{aligned}
&\ldots 00_b\, c_3\, 00_b\, c_1, \\
&\ldots 00_b\, c_4\, 00_b\, c_2, \\
&\ldots 00_b\, g_3\, 00_b\, g_1, \text{ and}
\end{aligned}$$

$$\dots 00_b\, g_4\, 00_b\, g_2,$$

which can be obtained by two shifts and 4 bitwise &-operations in total, if the ordering in the memory is $\dots c_4\, c_3\, c_2\, c_1$. Since the value of c_i is at most 2, a sevenfold summation of the first two displayed vectors leaves each component within its 4 bits (using any unsigned integer SIMD addition). Since the value of g_i is either 0 or 1, even a fifteen-fold summation is possible for the last two displayed vectors. Afterwards, the 4-bit values are extracted and further summed up. We call this method `Packed`.

The novel AVX512 command `_mm512_popcnt_epi64` might improve this approach, as it allows to count the number of bits being 1 in c_i, d_i and $e_i \mid f_i$, so that the number of products is counted that equal (i) 1^2 or 2^2, (ii) 2^2, and (iii) $1 \cdot 2$ or $1 \cdot 2$. This variant can be seen as a 2-bit analogue of the `Hamming2` algorithm, and will be implemented in future.

List of abbreviations

AVX: Advanced vector extensions

SIMD: Single instruction, multiple data

SNP: Single nucleotide polymorphism

SSE: Streaming SIMD extensions

SSSE: Supplemental streaming SIMD extensions

GRM: Genomic relationship matrix

Declarations

Ethics approval

Not applicable.

Consent for publication

Not applicable.

Availability of the data and materials

Not applicable.

Competing interests

The author declares that he has no competing interests.

Funding

None.

Author's contributions

Not applicable.

Acknowledgements

The author is grateful to Torsten Pook, Christopher Dörr and Alexander Freudenberg for hints and for comments.

References

1. VanRaden, P.M.: Efficient methods to compute genomic predictions. Journal of Dairy Science **91**(11), 4414–4423 (2008)
2. Walsh, B., Lynch, M.: Evolution and Selection of Quantitative Traits. Oxford University Press, Oxford (2018)
3. Rampazo Amadeu, R., Cellon, C., Olmestead, J.W., Franco Garcia, A., Resende Jr, M.F.R.: AGHmatrix: R package to construct relationship matrices for autotetraploid and diploid species. The Plant Genome **9**(3), 1–10 (2016)
4. Soerensen, P., Rohde, P.D., Soerensen, I.F.: qgg: An R package for Quantitative Genetic and Genomic analyses. http://psoerensen.github.io/qgg/ (2019)
5. Endelman, J.B.: Ridge regression and other kernels for genomic selection with R package rrBLUP. Plant Genome **4**, 250–255 (2011)
6. Granato, I., Fritsche-Neto, R.: snpReady: Preparing Genotypic Datasets in Order to Run Genomic Analysis. R package version 0.9.6. https://cran.r-project.org/web/packages/snpReady/index.html (2018)
7. Gogarten, S.M., Sofer, T., Chen, H., Yu, C., Brody, J.A., Thornton, T.A., Rice, K.M., Conomos, M.P.: Genetic association testing using the GENESIS R/Bioconductor package. Bioinformatics **35**, 5346–5348 (2019)
8. Yang, J., Lee, S.H., Goddard, M.E., Visscher, P.M.: GCTA: a tool for genome-wide complex trait analysis. The American Journal of Human Genetics **88**(1), 76–82 (2011)
9. Zheng, X., Levine, D., Shen, J., Gogarten, S.M., Laurie, W.B.S. C. and: A high-performance computing toolset for relatedness and principal component analysis of snp data. Bioinformatics **28**(24), 3326–3328 (2012). doi:10.1093/bioinformatics/bts606
10. Chang, C.C., Chow, C.C., Tellier, L.C.A.M., Vattikuti, S., Purcell, S.M., Lee, J.J.: Second-generation PLINK: rising to the challenge of larger and richer datasets. Gigascience **4**(1), 7 (2015)
11. Schlather, M., Erbe, M., Skene, F., Freudenberg, A.: miraculix: Algebraic and Statistical Functions for Genetics. (2019). R package version 0.9.15. https://github.com/schlather/miraculix
12. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2019). R Foundation for Statistical Computing. https://www.R-project.org/
13. Ben-Haim, G.: Practical Intel(R) AVX Optimization on 2nd generation Intel(R) Core$^{\mathrm{TM}}$ Processors. https://software.intel.com/en-us/articles/practical-intel-avx-optimization-on-2nd-generation-intel-core-processors (2012)
14. Purcell, S., Neale, B., Todd-Brown, K., Thomas, L., Ferreira, M.A.R., Bender, D., Maller, J., Sklar, P., de Bakker, P.I.W., Daly, M.J., Sham, P.C.: PLINK: a toolset for whole-genome association and population-based linkage analysis. American Journal of Human Genetics **81**(3), 559–575 (2007)
15. Dalke, A.: Simple benchmark harness for different popcount implementations. http://dalkescientific.com/writings/diary/popcnt.cpp (2011)
16. Dalke, A.: Update: Faster population counts. http://www.dalkescientific.com/writings/diary/archive/2011/11/02/faster_popcount_update.html (2011)