**Supplementary Materials for:**

**OCMA: Fast, memory-efficient factorization** **of prohibitively large relationship matrices**

Zhi Xiong1, Qingrun Zhang2,3, Alexander Platt4, Wenyuan Liao8, Xinghua Shi5, Gustavo de los Campos6, and Quan Long2,7,8,9

1. Department of Computer Science, Shantou University, China
2. Department of Biochemistry and Molecular Biology, University of Calgary, Canada
3. Annie Charbonneau Cancer Institute, University of Calgary, Canada
4. Center for Computational Genetics and Genomics, Temple University, USA
5. Department of Bioinformatics and Genomics, University of North Carolina at Charlotte, USA
6. Department of Epidemiology & Biostatistics, Statistics & Probability and Institute for Quantitative Health Science and Engineering, Michigan State University, USA
7. Department of Medical Genetics, University of Calgary, Canada
8. Department of Mathematics and Statistics, University of Calgary, Canada
9. Alberta Children Hospital Research Institute, University of Calgary, Canada

Correspondence should be addressed to Q.L. (quan.long@ucalgary.ca)

**Keywords**: Eigen decomposition, Singular value decomposition, Genetic matrices, Memory virtualization, Gene mapping, Genotype-based phenotype prediction, Genomic selection.

**Supplementary Note I: Details of the implementation**

Memory-mapped file I/O (mmap)

In computer science, Memory-Map or *mmap* is a system call that enables “memory virtualization”, which stores a file or device on the disk and access it as though they are in the memory. Technically, it maps the virtual address space of a computing process to the corresponding physical address in the disk, which allows mapping files or devices into memory [1].

As depicted in **Figure 1** of the main text, when the files have been mapped to the virtual address of the computing process by *mmap*, the program can directly read and write the files using the virtual address (as though they were just resided in the memory) without frequently invoking the system-level read and write operations (as people usually do to access the files on the disk).

At the beginning, the target files do not have to be loaded into the main memory fully. (As a result there won’t be a long waiting time of “loading files” before any analysis can be done.) The operating system will load the part that is accessed by the computing process on-the-fly in a “lazy” manner. When the memory is insufficient, the relatively inactive part will be swapped out of the memory. So *mmap* can be used to map huge files with limited memory. The operating system of the computer will automatically manage data exchange. As a result, from the perspective of the programmers and the users, the complicated data exchange is invisible, a property called “transparency” in computer science.

The prototype of mmap can be spelled as follows:

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset)

In the Linux and Mac OS systems, we use mmap to directly map an opened file into the memory. In the Windows system, we sequentially use three functions, i.e., CreateFile, CreateFileMapping, and MapViewOfFile, to map the file into the memory. After this has been achieved, the rest of the programs can process the file as if it is in the memory.

Solutions for eigen-decomposition and singular-value decomposition

Intel Math Kernel Library (Intel MKL) [2] is a set of highly optimized mathematical libraries for scientific computing. We selected the function “ssyevd” [3] to solve the eigenvalues and eigenvectors of the GRM, and “sgesdd” [4] to carry out singular value decompositions (an economy-sized decomposition). The designs of the two procedures are quite similar. Here only the design of eigen-decomposition is described.

The ssyevd routine computes all eigenvalues and eigenvectors of a real symmetric matrix using divide and conquer algorithm. The prototype is spelled out as below:

void ssyevd( const char\* jobz, const char\* uplo, const MKL\_INT\* n, float\* a,

const MKL\_INT\* lda, float\* w, float\* work, const MKL\_INT\* lwork, MKL\_INT\* iwork, const MKL\_INT\* liwork, MKL\_INT\* info );

Regularly (without the address mapping between disk and memory), this function needs to apply for memory to store 4 arrays with various sizes. They are “a”, “w”, “work” and “iwork”. Their theoretical sizes in terms of the dimension of a matrix are displayed in **Supplementary Table S1**. Their expected actual sizes with regard to the dimensions of the GRM is explained in **Supplementary Table S2** and **S3.** Two of these four arrays, “a” and “work” are very large. Based on the availability of memory, users can specify whether to use memory- or disk-based methods of OCMA to handle these large workspaces. The other two arrays are relatively small and in general sized at the level of MB; therefore they are stored in the memory.

Configuration of the test machine: We used a desktop personal computer with Intel I7 CPU, 24GB memory and 250GB disk to test our software and carry out the comparisons. The detailed parameters of this machine are listed in **Supplementary Table S4**.

**Supplementary Note II: Details of estimating the runtime and memory usage of GCTA:**

GCTA doesn’t provide a function that carries out the matrix-decomposition only. As an alternative we used the GREML function as an approximation. Here is the approaches we adopted to estimate the memory usage and computational time.

*Memory use*: The GCTA website offered a formula to calculate the memory usage of GREML analyses:

“*It involves a number of n x n matrices, e.g. GRM, variance-covariance V matrix, the projection P matrix and temporary matrices for V inverse calculation. Total memory usage ~= (t + 4) \* n \* n \* 8 bytes, where t is the number of genetic components (i.e. the number of GRMs) fitted in the model*.”

In this computation, t (the number of GRMs) is 1. We assume that GCTA needs at least one other *n* x *n* matrix (i.e., a V inverse matrix) in the process of decomposition. Therefore took the *ad hoc* formula: *memory usage ~= 2 \* n \* n \* 8 bytes.* Please note this is a minimal low-end bound that is in favor of counting less memory use.

*Computational time*: The GREML estimation involves many matrix operations including matrix inverse, Cholesky decomposition, LU factorization, and several other matrix multiplications. When matrix inverse, which consumes the same magnitude of runtime of an eigen-decomposition, is done, all the other operations takes a computing time of the magnitude of *O(N2).* So the matrix inverse is the only operation that needs a computing time of the magnitude of *O(N3).* In light of this, we assume that half of the overall computational time to estimate GREML is consumed by the matrix inverse, which is comparable to our eigen-decomposition.

We do acknowledge that our approximation of the computational time may be imprecise as the result of a very large constant multiplies *N2* could be larger than *N3*. However, this is unlikely for an extremely large *N.* Also, the main message here is to demonstrate that the algorithm we used, i.e., Intel MKL “ssyevd/dsyevd”, is the best available implementation for eigen-decomposition of very large matrices. This is actually consistent to our experience of carrying out numeric computing for a decade and a general consensus in the field.

**Supplementary Notes III: Details of running the benchmarks**

Specific functions we have used in building OCMA and testing other tools:

Intel MKL version: l\_mkl\_2018.0.128 for Linux and w\_mkl\_2018.1.156 for Windows.

MATLAB: the function “eig” is used for eigen-decomposition and “svd” is used for singular value decomposition.

GCTA: the function “greml” is use for the GREML estimation.

OpenBLAS: we used the function “ssyevd” (the same as the one we adopted in Intel MKL).

Compiler used in Windows: Microsoft C/C++ Compiler (Visual Studio 2013).

Compiler used in Linux: GNU C Compiler (GCC 4.8.5).

Mmap under Windows system: In Windows system, the memory mapping (i.e., mmap in Linux) is structured by running three functions, “CreatFile”, “CreateFileMapping”, and “MapViewOfFile” sequentially. For simplicity, we still refer to this process as mmap in throughout this paper.

OCMA: single precision floating numbers are used in all the calculation by electing the option “single”. The function “eigen” and “singular” are used to run eigen-decomposition and SVD.

Swap usage:

In our evaluations, we used the defaults setting for both Linux and Windows. The maximal space for swap is 1.5 time of the physical memory in Windows and Linux. It appears that Windows is more efficient than Linux for both swap and mmap. In practice, the swap usage is subject to other processes in the same machine, therefore it is unclear for us exactly how much swap spaces are used by the processes carrying out matrices factorization.

Generation of the GRM

The genomic relationship matrix was the same as the one used in Kim et al. (Genetics, 2017). Briefly, genomic relationships were computed using genotypes from white-British individuals included in the interim release of the UK-Biobank. SNPs were from the Affymetrix UK BiLEVE Axiom and Affymetrix UK Biobank Axiom arrays. The number of SNPs available was 847,441. After filtering (which consisted of retaining SNPs with minor-allele-frequency greater than 0.1% and call rate greater than 97%) a total of 589,028 were available to compute genomic relationships. Genomic relationships were computed using the getG() function of the BGData package using options that center and scale genotypes. Thus, the genomic relationship between the ith and jth individual is computed as where is the genotype (0/1/2) of the ith individual at the kth loci, is the frequency of the allele coded as 1 at the kth loci and p=589,028 is the number of SNPs used. We computed the genomic relationship in MSU High Performance Computing clusters. Blocks of the genomic relationship matrix were computed in separate nodes using methods described in BGData users’ manual (https://cran.r-project.org/web/packages/BGData/index.html). Only pairs of individuals with (n=102,643) were retained.

Generation of the genotype matrix

As we need to demonstrate the use of OCMA for SVD calculation on a genotype matrix with very large sample size close to 1 million but do not have a dataset at this level, we simulated a genotype matrix based on the template of 1,000 Genomes Project. Based on the average divergence between any two individuals in the 1,000 Genomes Project and the allele frequencies, we simulated new individuals by mutating the existing individuals. **Supplementary Tables**

**Table S1**. **Four arrays and their sizes used in Intel MKL |** The integer “*N*” is the length of the matrix, i.e., the sample size in a biobank. The array “a” stores the input matrix and the output eigenvectors. The array “w” stores the output eigenvalues. The array “work” is a workspace storing float numbers with variable length depending on the value of *N*. The array “iwork” is a workspace storing integer numbers with variable length depending on the value of *N*. The function “ssyevd” also provides a method to calculate the minimal sizes of these two workspaces, presented in **Table S2**. The actual spaces needed by these four arrays are presented in **Table S3**.

|  |  |  |  |
| --- | --- | --- | --- |
| Array | Content | Size | Space size (B) |
| *a* | matrix (in) / eigenvectors of the matrix (out) | *N*\**N* | *N*\**N*\*sizeof(float) |
| *w* | eigenvalues of the matrix | *N* | *N*\*sizeof(float) |
| *work* | float workspace | *lwork* | *lwork*\*sizeof(float) |
| *iwork* | integer workspace | *liwork* | *liwork*\*sizeof(MKL\_INT) |

**Table S2. The required minimal size of lwork and liwork**

|  |  |  |
| --- | --- | --- |
| *N* | *lwork* | *liwork* |
| 20000 | 800,120,064 (0.75G) | 100003 (0.10M) |
| 40000 | 3,200,240,128 (2.98G) | 200003 (0.19M) |
| 60000 | 7,200,360,448 (6.71G) | 300003 (0.29M) |
| 80000 | 12,800,480,256 (11.92G) | 400003 (0.38M) |
| 100000 | 20,000,600,064 (18.63G) | 500003 (0.48M) |

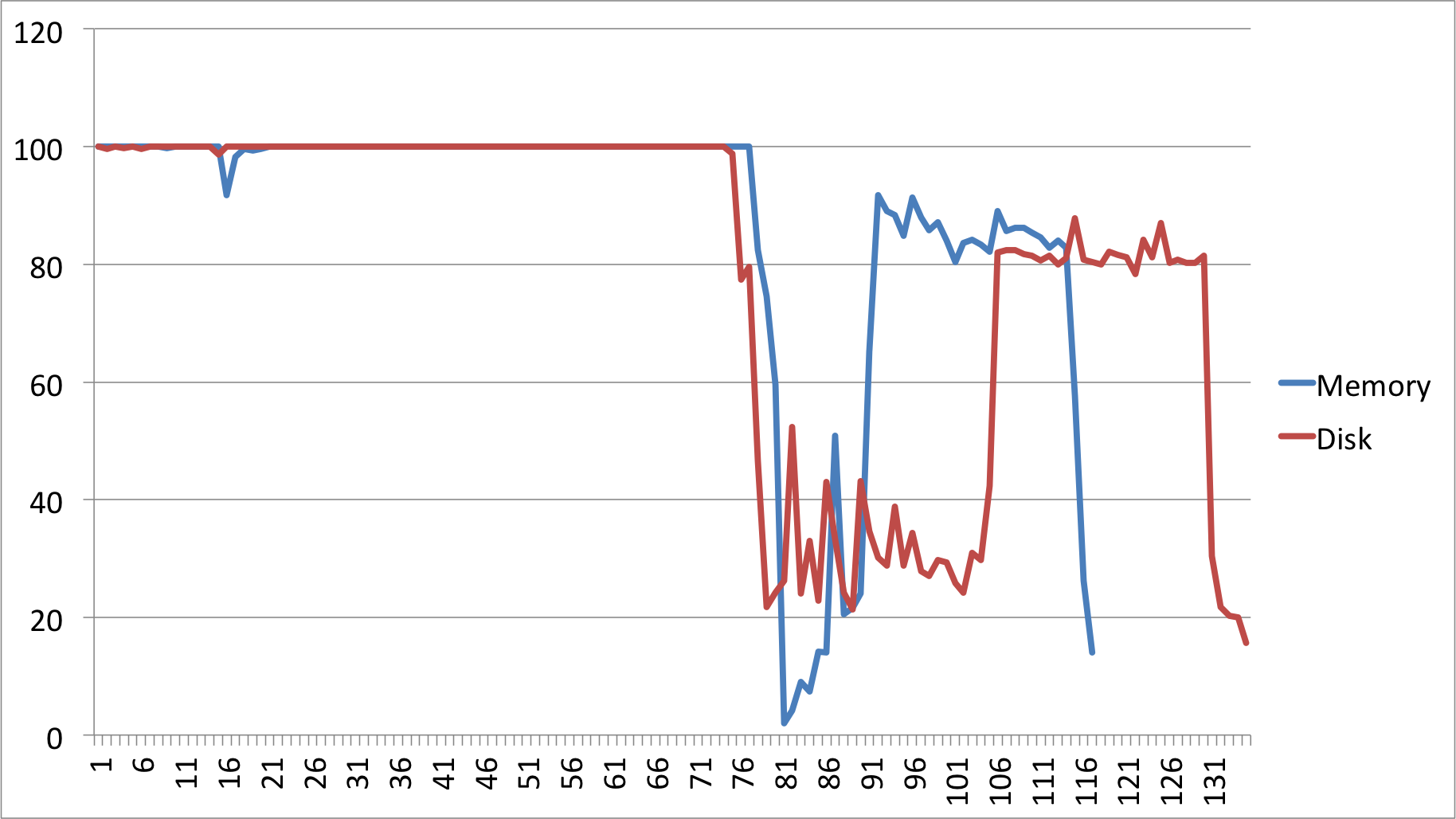
**Table S3. Spaces (in memory or disk) needed for the four arrays**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* | *a* (GB) | *w* (MB) | *work* (GB) | *iwork* (MB) |
| 20000 | 1.49 | 0.08 | 2.98 | 0.76 |
| 40000 | 5.96 | 0.15 | 11.92 | 1.53 |
| 60000 | 13.41 | 0.23 | 26.82 | 2.29 |
| 80000 | 23.84 | 0.31 | 47.69 | 3.05 |
| 100000 | 37.25 | 0.38 | 74.51 | 3.81 |

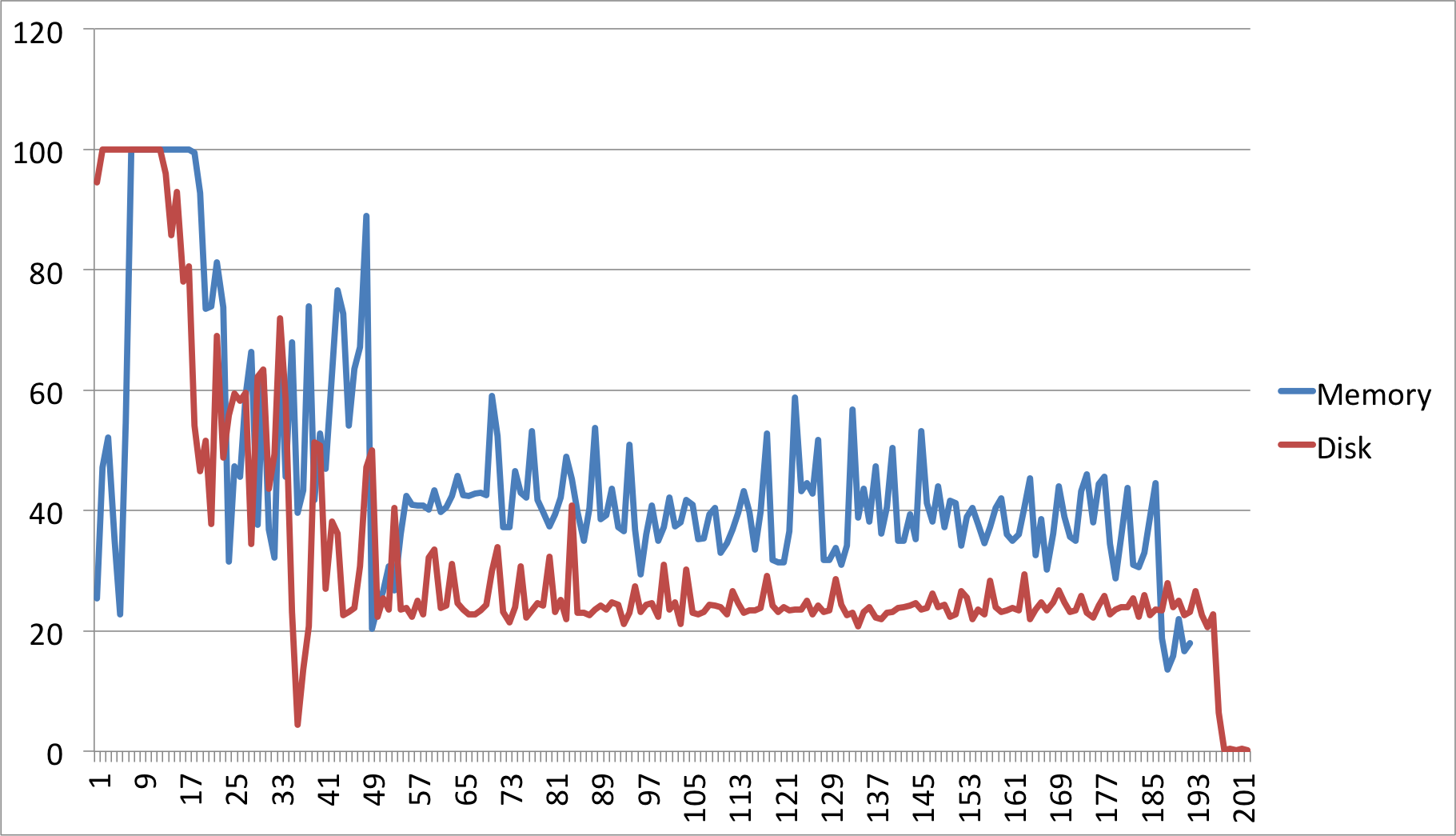
**Table S4. Configuration of the desktop computer used to test the software**

|  |  |
| --- | --- |
| Item | Configuration |
| CPU | Intel Core i7-6700 CPU. 4 cores |
| Memory | 24 GB |
| Disk | Samsung SSD 850 EVO 250GB |
| Operating System | CentOS Linux release 7.3.1611  Windows 7 |

**Supplementary Figures**



**Figure S1 | CPU usage of Eigen-decomposition of OCMA using memory- and disk-based methods**. X-axis is the time elapsed (minutes); y-axis is the CPU usage (%). Sample size *N* = 60,000. The test is carried out based on the Windows computer.



**Figure S2 | CPU usage of Singular Value Decomposition of OCMA using memory- and disk-based methods**. X-axis is the time elapsed (minutes); y-axis is the CPU usage (%). Number of markers *M* = 6,000. The test is carried out based on the Windows computer.

[1] https://en.wikipedia.org/wiki/Mmap

[2] https://software.intel.com/en-us/mkl

[3] https://software.intel.com/en-us/mkl-developer-reference-fortran-syevd

[4] https://software.intel.com/en-us/mkl-developer-reference-fortran-gesdd