Available online at http://scik.org
J. Math. Comput. Sci. 10 (2020), No. 3, 497-506
https://doi.org/10.28919/jmcs/4457
ISSN: 1927-5307

MEMORY-EFFICIENT SELF-CROSS-PRODUCT FOR LARGE MATRICES USING R AND PYTHON

MOHAMMAD ALI NILFOROOSHAN*

Livestock Improvement Corporation, Private Bag 3016, Hamilton 3240, New Zealand

Abstract. In many quantitative studies, calculations of matrix self-cross-products ($\mathbf{B}'\mathbf{B}$) are needed, where $\mathbf{B}$ is any matrix of interest. For matrix $\mathbf{B}$ with many number of rows, there might be memory limitations in storing $\mathbf{B}'\mathbf{B}$. Also, calculating $\mathbf{B}'\mathbf{B}$ has a computational complexity of m^2n, for $\mathbf{B}$ with n and m number of rows and columns, respectively. Because $\mathbf{B}'\mathbf{B}$ is symmetric, almost half of the calculations and the memory usage are redundant. The half-matrix multiplication algorithm (HMMA) was introduced, which creates $\mathbf{B}'\mathbf{B}$ upper-triangular. Matrix multiplication functions `%*%` and `crossprod` in R, `numpy.dot` in Python, and user-defined HMMA functions `hmma_r` and `hmma_py` in R and Python were compared, for matrix $\mathbf{B}$ with 40,000 real numbers, and various dimensions. Runtime of $\mathbf{B}'\mathbf{B}$ calculation was less than a second when $\mathbf{B}$ had more than 4 rows. The longest runtime was for $\mathbf{B}$ with 1 row and `crossprod` (21.3 sec), and then `numpy.dot` (9.7 sec). Considering $\mathbf{B}$ with 4 or less number of rows, `hmma_py`, `%*%`, and `hmma_r` ranked 1 to 3 for the shortest runtime. The memory usage of a (40,000 $\times$ 40,000) $\mathbf{B}'\mathbf{B}$ was 12.8 Gb, and the main advantage of HMMA was reducing it to the half.

Keywords: matrix; cross-product; symmetric; computational complexity; memory usage; runtime; animal model.

2010 AMS Subject Classification: 17D92.

*Corresponding author

E-mail address: mohammad.nilforooshan@lic.co.nz

Received January 5, 2020

1. INTRODUCTION

In many fields of quantitative science, the cross-product of a matrix to itself (transpose of a matrix multiplied to the matrix) is needed. Quantitative Genetic is one of them, in which, animal models have been widely used in livestock genetic evaluations. There are numerous types of animal models, some of them explained by Mrode [1]. All animal models are based on the best linear unbiased prediction (BLUP) methodology developed by Henderson [2]. As an example, a repeatability animal model is [1]:

$$\mathbf{y} = \mathbf{Xb} + \mathbf{Za} + \mathbf{Wpe} + \mathbf{e}, \tag{1}$$

where **y**, **b**, **a**, **pe**, and **e** are the vectors of observations, fixed effects, random animal genetic effects, random permanent environmental effect, and random residuals, and **X**, **Z**, **W** are incidence matrices relating **y** to **b**, **a**, **pe**, respectively. In matrix notations, Eq. (1) is written as [1]:

$$\begin{bmatrix} \mathbf{X}'\mathbf{X} & \mathbf{X}'\mathbf{Z} & \mathbf{X}'\mathbf{W} \\ \mathbf{Z}'\mathbf{X} & \mathbf{Z}'\mathbf{Z}+\mathbf{A}^{-1}\alpha_1 & \mathbf{Z}'\mathbf{W} \\ \mathbf{W}'\mathbf{X} & \mathbf{W}'\mathbf{Z} & \mathbf{W}'\mathbf{W}+\mathbf{I}\alpha_2 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{b}} \\ \hat{\mathbf{a}} \\ \hat{\mathbf{pe}} \end{bmatrix} = \begin{bmatrix} \mathbf{X}'\mathbf{y} \\ \mathbf{Z}'\mathbf{y} \\ \mathbf{W}'\mathbf{y} \end{bmatrix}, \tag{2}$$

where **A** is the pedigree relationship matrix, **I** is an identity matrix with the order of the total number of animals in the pedigree, $\alpha_1 = \sigma_e^2/\sigma_a^2$, $\alpha_2 = \sigma_e^2/\sigma_{pe}^2$, σ_a^2, σ_{pe}^2, and σ_e^2 are the additive genetic, permanent environment, and residual variances, respectively. Given the left-hand-side matrix and the right-hand-side vector, $\hat{\mathbf{b}}$, $\hat{\mathbf{a}}$, and $\hat{\mathbf{pe}}$ vectors are being predicted. There are matrix self-cross-products ($\mathbf{X}'\mathbf{X}$, $\mathbf{Z}'\mathbf{Z}$, and $\mathbf{W}'\mathbf{W}$) on the diagonal blocks of the left-hand-side matrix. These square matrices have sizes equal to the total number of levels for fixed effects, number of animals in the pedigree, and the number of animals with observed phenotypes, respectively. There might be millions of animals in the pedigree.

If various residual variances are associated with **y**, the equation system (2) is changed to [1]:

$$(3)\qquad \begin{bmatrix} \mathbf{X}'\mathbf{R}^{-1}\mathbf{X} & \mathbf{X}'\mathbf{R}^{-1}\mathbf{Z} & \mathbf{X}'\mathbf{R}^{-1}\mathbf{W} \\ \mathbf{Z}'\mathbf{R}^{-1}\mathbf{X} & \mathbf{Z}'\mathbf{R}^{-1}\mathbf{Z}+\mathbf{A}^{-1}\sigma_a^{-2} & \mathbf{Z}'\mathbf{R}^{-1}\mathbf{W} \\ \mathbf{W}'\mathbf{R}^{-1}\mathbf{X} & \mathbf{W}'\mathbf{R}^{-1}\mathbf{Z} & \mathbf{W}'\mathbf{R}^{-1}\mathbf{W}+\mathbf{I}\sigma_{pe}^{-2} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{b}} \\ \hat{\mathbf{a}} \\ \hat{\mathbf{pe}} \end{bmatrix} = \begin{bmatrix} \mathbf{X}'\mathbf{R}^{-1}\mathbf{y} \\ \mathbf{Z}'\mathbf{R}^{-1}\mathbf{y} \\ \mathbf{W}'\mathbf{R}^{-1}\mathbf{y} \end{bmatrix},$$

where $\mathbf{R}$ is the diagonal matrix of residual variances. Matrix self-cross-products do still exist. For example, $\mathbf{X}'\mathbf{R}^{-1}\mathbf{X} = (\mathbf{R}^{-.5}\mathbf{X})'\mathbf{R}^{-.5}\mathbf{X}$. Also, $\mathbf{V} = [\mathbf{X}\ \mathbf{Z}\ \mathbf{W}]$:

$$(4)\qquad \left[\mathbf{V}'\mathbf{R}^{-1}\mathbf{V}+\begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{-1}\sigma_a^{-2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}\sigma_{pe}^{-2} \end{bmatrix}\right] \begin{bmatrix} \hat{\mathbf{b}} \\ \hat{\mathbf{a}} \\ \hat{\mathbf{pe}} \end{bmatrix} = \mathbf{V}'\mathbf{R}^{-1}\mathbf{y}.$$

The genomic relationship matrix ($\mathbf{G}$), which is used instead of $\mathbf{A}$ in genomic-BLUP [3] is also a matrix self-cross-product. There are several forms of $\mathbf{G}$ (e.g., VanRaden [3], Yang et al. [4]). For example, in one of the methods introduced by VanRaden [3], $\mathbf{G} = \mathbf{L}'\mathbf{L}$, where $\mathbf{L} = \mathbf{Z}'/\sqrt{2\sum p_i(1-p_i)}$, $\mathbf{Z}$ is a centered incidence matrix of SNP genotypes, and p_i is the allele frequency at loci i.

A matrix self-cross-product is always symmetric. As a result, $(m-1)m/2$ of the calculations and the memory usage are redundant, where the dimension of the cross-product is $m \times m$. The aim of this study was finding an algorithm for obtaining the upper triangular of a matrix self-cross-product in the shortage of memory, and benchmarking runtime and memory usage with high level programming languages, R and Python. Whereas, there are advanced matrix multiplication algorithms in mathematical libraries of low-level languages such Fortran and C, it was of interest exploring how programmers in high-level languages such as R and Python can perform these operations, when due to the size of the problem, memory usage is a limitation.

2. MATERIALS

A vector of 40,000 random real numbers, sampled from a uniform distribution, ranged from 0 to 100, was used to form matrix $\mathbf{B}$, from which $\mathbf{B}'\mathbf{B}$ was going to be calculated. Various $\mathbf{B}$ were formed with different dimensions, and the number of rows being any integer from 1 to

40,000, with a remainder of 0 with 40,000. That means, there were 35 values for the number of rows, where the first 6 values were 1, 2, 4, 5, 8, and 10.

Benchmarking were run on a t3.xlarge Amazon EC2 instance, which features Intel Xeon Platinum 8000 series processor with Turbo CPU clock speed of up to 3.1 Ghz, 4 virtual CPUs, 16 Gb of RAM, and solid-state drive volume type. Softwares R 3.4.4 [5], Python 3.6.7 [6], and the Python library numpy 1.16.2 [7] were installed on an Ubuntu Server 18.04 LTS-64-bit (x86) operating system.

3. METHODS

The computational complexity of $\mathbf{B}'\mathbf{B}$ is m^2n, where n and m are the number of rows and columns of $\mathbf{B}$, respectively. The dimension of $\mathbf{B}'\mathbf{B}$ is $m \times m$, and it is symmetric. Thus, not all the multiplications are necessary. Introducing the half-matrix multiplication algorithm (HMMA), to calculate the upper triangular of $\mathbf{B}'\mathbf{B}$:

$$\text{FOR } j \text{ IN } (1{:}m)\text{: } \mathbf{B}'\mathbf{B}[j,\ j..m] = \mathbf{B}'[j,]\mathbf{B}[,j..m]$$

Examples for a (3×4) $\mathbf{B}$ matrix are:

```
B = matrix(0:11, nrow=3)
BtB = list()
for(j in 1:ncol(B)) BtB[[j]] = B[,j] %*% B[,j:ncol(B)]
```

in R, and the following in Python:

```
B = numpy.array(range(12))
B.shape = (3, B.size//3)
BtB = []
  for j in range(B.shape[1]):
    BtB.append(B[:,j].dot(B[:,j:]))
```

Using HMMA, the computational complexity is reduced to $nm(m+1)/2 + \sum_{k=1}^{m} l_k$, where l is the loop cost, decreasing by increasing the iteration number. Fig. 1 shows the reduction in the computational complexity of $\mathbf{B}'\mathbf{B}$ ($nm(m + 1)/2 - m^2n$) by using HMMA, excluding the loop cost, with the number of rows (n) and the number of columns (m) of $\mathbf{B}$ being variable from 1 to 10.

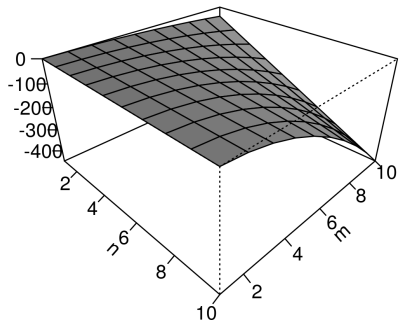

FIGURE 1. The effect of n and m on $nm(m+1)/2 - m^2n$.

However, additional costs are involved in forming the full matrix after HMMA. The steps of retrieving the full $\mathbf{B}'\mathbf{B}$ matrix from the list object `BtB` obtained from HMMA are described in Appendix A (R function `list2mat`). However, forming the full $\mathbf{B}'\mathbf{B}$ matrix might not always be necessary. As for multiplication with other vector or matrix, only rows of $\mathbf{B}'\mathbf{B}$ need to be retrieved, one after another. R function `get_rowcol` (Appendix B) extracts the ith row/column of $\mathbf{B}'\mathbf{B}$ from the `BtB` list. Given $\mathbf{B}$ with 40,000 elements and various dimensions, runtime in seconds, and memory usage in bytes were compared across `%*%`, `crossprod`, and `hmma_r` (HMMA, Appendix C) functions in R, and `numpy.dot`, and `hmma_py` (HMMA, Appendix D) functions in Python. Runtime was averaged over 4 reiterations. Using `%*%` or `crossprod` inside the `hmma_r` function (the 4th line of the function (Appendix C)) produced similar timimg results. Therefore, results for `hmma_r` with `%*%` inside are presented.

4. RESULTS AND DISCUSSION

4.1. Runtime. Runtime for $\mathbf{B}'\mathbf{B}$ calculation or its upper triangular (HMMA) were compared for the 3 functions in R (`%*%`, `crossprod`, `hmma_r`) and the 2 functions in Python (`numpy.dot`, `hmma_py`), for $\mathbf{B}$ with 40,000 elements and various dimensions. The results for $\mathbf{B}$'s number of rows from 1 to 10 are presented in Fig. 2. Runtime for $\mathbf{B}$ with more number of rows was very short, and provided in the data repository (Data Availability).

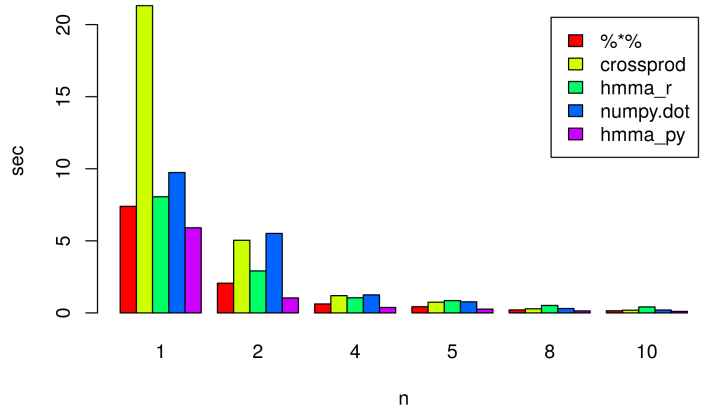

FIGURE 2. Runtime of $\mathbf{B}'\mathbf{B}$, with different $\mathbf{B}$ of 40,000 elements and different number of rows (n).

As the number of $\mathbf{B}$'s rows (n) increased, the dimension of $\mathbf{B}'\mathbf{B}$ and its computational complexity decreased. `hmma_py` showed the least runtime, and `crossprod` showed a longer runtime compared to its R-native equivalent `%*%`. Though, the time difference was small at n = 20, `crossprod`'s runtime was shorter than `%*%`'s runtime (result not shown, but available in the data repository). At n = 1, `crossprod` showed the longest runtime. However, at $n >$ 1, `numpy.dot`'s runtime was longer. Loop cost increased, and the computational complexity decreased by increasing n. As a result HMMA's benefit decreased. At $n \geq 5$, `hmma_r` showed the longest runtime. However, at that point, all $\mathbf{B}'\mathbf{B}$ calculations were taking less than a second time. Therefore, parallel processing to reduce the runtime was not necessary.

4.2. Memory usage. Memory usage of $\mathbf{B}'\mathbf{B}$ or its upper triangular (HMMA) were compared for the 3 functions in R (`%*%`, `crossprod`, `hmma_r`) and the 2 functions in Python (`numpy.dot`, `hmma_py`), for $\mathbf{B}$ with 40,000 elements and various dimensions. The results for $\mathbf{B}$'s number of rows from 1 to 10 are presented in Fig. 3. Memory usage of $\mathbf{B}'\mathbf{B}$ for $\mathbf{B}$ with more number of rows are provided in the data repository.

Memory usage of $\mathbf{B}'\mathbf{B}$ by `hmma_r` and `hmma_py` were almost half of the memory usage by the other functions. The memory usage by `%*%` and `crossprod` were equal, and the memory usage by `numpy.dat` was 88 bytes less than the memory usage by `crossprod`. Increasing

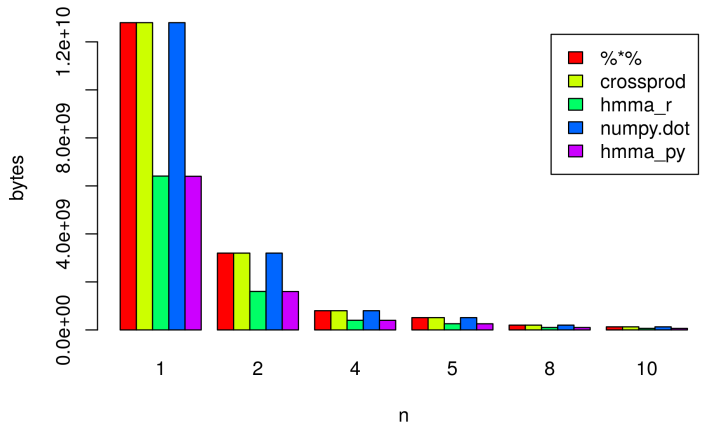

FIGURE 3. Memory size of $\mathbf{B}'\mathbf{B}$, with different $\mathbf{B}$ of 40,000 elements and different number of rows (n).

n from 1 to 40,000 ($\mathbf{B}'\mathbf{B}$ dimension from 40,000 $\times$ 40,000 to 1 $\times$ 1), the memory usage of $\mathbf{B}'\mathbf{B}$ reduced from 12,800,000,200 to 208 bytes for `crossprod` and `%*%`, 6,408,480,288 to 256 bytes for `hmma_r`, and 6,400,160,096 to 104 bytes for `hmma_py`.

Testing the memory limit, a (1 $\times$ 46,000) $\mathbf{B}$ was created to form a (46,000 $\times$ 46,000) $\mathbf{B}'\mathbf{B}$. Both `%*%` and `crossprod` failed, and the system returned the message "`Error: cannot allocate vector of size 15.8 Gb`". On the other hand, `hmma_r` successfully built $\mathbf{B}'\mathbf{B}$ with 7.9 Gb memory usage. `numpy.py` failed and returned a "`MemoryError`" message. Also, `hmma_py` failed with the following message:

```
    BtB = numpy.concatenate(BtB) # to get the correct memory usage
MemoryError
```

Excluding `BtB = numpy.concatenate(BtB)` from `numpy.dot`, it successfully created $\mathbf{B}'\mathbf{B}$. This line of code was not necessary, but to get the correct estimate of the memory usage. It seems that the problem was keeping $46{,}000^2$ real numbers in a single array rather than multiple arrays in a list. To get the correct memory usage without this line of code, `sys.getsizeof` (in the last line of `hmma_py` (Appendix D)) should iterate over the arrays in the `BtB` list and sum their sizes.

Storing float numbers with single-precision allows saving memory usage. R does not have the ability to work with single-precision [8]. Thus, R stores numeric matrices with double-precision, which takes double amount of memory (64-bit vs. 32-bit floats). Single-precision accommodates a precision of approximately 7 decimal digits, and double-precision with a precision of approximately 16 decimal digits [9]. There is a trade-off between memory usage and the accuracy, but in most of the tasks, high accuracy in long decimal points is not required, and single-precision can be satisfactory. The `float` package [8] extends R's linear algebra facilities to include single-precision (float) data. Python has a rich family of data types, arrays can save floats in single and double precision, and libraries like ctypes [10] and numpy [7] provide the possibility of storing floats in single- and double-precision.

5. CONCLUSIONS

Even though, HMMA does almost half ((1 + 1/m)/2, where m is the matrix's number of columns) of the matrix self-cross-product multiplications, it did not necessarily reduce the computational time. The reason was the additional loop cost. However, runtime was not a constraint, because all the calculations went reasonably fast. In cases where the memory usage is a constraint, HMMA can be used to reduce the memory usage of $\mathbf{B}'\mathbf{B}$ by (1 + 1/m)/2, and maximum vector length by 1/m.

CONFLICT OF INTERESTS

The author(s) declare that there is no conflict of interests.

REFERENCES

[1] R. A. Mrode, Linear Models for the Prediction of Animal Breeding Values (2nd ed.), CABI, Oxfordshire, UK (2005), pp.71–81.

[2] C. R. Henderson, Sire evaluation and genetic trends, J. Anim. Sci. 1973 (Symposium) (1973), 10–41.

[3] P. M. VanRaden, Efficient methods to compute genomic predictions, J. Dairy Sci. 91(11) (2008), 4414–4423.

[4] J. Yang, B. Benyamin, B. P. McEvoy, S. Gordon, A. K. Henders, D. R. Nyholt, P. A. Madden, et al., Common SNPs explain a large proportion of the heritability for human height, Nat. Genet. 42 (2010), 565–569.

[5] R Core Team, The R Project for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, `https://www.R-project.org` (2018). [Online; accessed 28-June-2019]

[6] Python Software Foundation, Python 3.6 documentation – The Python standard library, https://docs.python.org/3.6/library/index.html (2019). [Online; accessed 28-June-2019]

[7] P. F. Dubois, K. Hinsen, J. Hugunin, Numerical Python, Comput. Phys. 10 (1996), Article ID 262.

[8] D. Schmidt, Introducing the float package: 32-Bit Floats for R (Version 0.2-3), https://cran.rstudio.com/web/packages/float/vignettes/float.pdf (2019). [Online; accessed 28-June-2019]

[9] Wikipedia contributors, Floating-point arithmetic, https://en.wikipedia.org/wiki/Floating-point_arithmetic (2019). [Online; accessed 28-June-2019]

[10] Python Software Foundation, ctypes – A foreign function library for Python, https://docs.python.org/3.6/library/ctypes.html (2019). [Online; accessed 28-June-2019]

Data Availability: The supporting data is available at the Mendeley Data repository: DOI:10.17632/vk8vy7ghnf.1

APPENDICES

Appendix A. `list2mat` function in R:

```
list2mat = function(BtB) {
  dimmat = (sqrt(1+8*length(unlist(BtB)))-1)/2
  mat = matrix(0, nrow=dimmat, ncol=dimmat)
  k = 0
  for(i in 1:dimmat)
  {
    mat[i, i:dimmat] = BtB[[i]]
    k = k + dimmat - i
  }
  mat = mat + t(mat)
  diag(mat) = diag(mat)/2
  return(mat)
}
```

Appendix B. `get_rowcol` function in R:

```
get_rowcol = function(BtB, th) {
  if(th < 1 | th > length(BtB)) stop("Invalid row/column")
```

```
  rc = c()
  if(th > 1)
  {
    for(i in 1:(th-1)) rc = c(rc, BtB[[i]][th-i+1])
    rc = c(rc, BtB[[th]])
  } else {
    rc = BtB[[1]]
  }
  return(rc)
}
```

Appendix C. `hmma_r` function in R:

```
hmma_r = function(B) {
  start_time = Sys.time()
  BtB = list()
    for(i in 1:ncol(B)) BtB[[i]] = B[,i] %*% B[,i:ncol(B)]
  print(paste(round(as.numeric(Sys.time()-start_time, units="secs"), 3),
              object.size(BtB), nrow(B), ncol(B), sep=","))
}
```

Appendix D. `hmma_py` function in Python:

```
def hmma_py(B):
  start_time = time.time()
  BtB = []
  for i in range(B.shape[1]):
    BtB.append(B[:,i].dot(B[:,i:]))
  BtB = numpy.concatenate(BtB) # to get the correct memory usage
  print("%s,%d,%d,%d" %(round(time.time()-start_time, 3), \
  sys.getsizeof(BtB), B.shape[0], B.shape[1]))
```