Optimal Mapped Matrix Multiplication in GNU Octave

Kyle Haptonstall
khaptonstall@monmouthcollege.edu

James Logan Mayfield, Ph.D.
jmayfield@monmouthcollege.edu

January 8, 2015

Abstract

The aim of this paper is to compare three ways of achieving mapped matrix multiplication in GNU Octave [1]. There are many way to do this, and three different ways will be focused on here. The goal is to find the right combination of data structures and algorithms for as both the size of the matrices and the number of matrices grow. It is shown that vectorized techniques speed up computation for small matrix multiplication, but overall storing sets of matrices in cell arrays and using loops to achieve mapped matrix multiplication proves to be computationally faster overall.

Introduction

This paper presents and analyses three ways to do mapped matrix multiplication. Mapped matrix multiplication consists of multiplying each element of a set of matrices, each with a single matrix. Currently, Octave does not have a built in function to do mapped matrix multiplication. Users must therefore develop their own time efficient mapped matrix multiplication function.

Octave common practice is to use vectorization for computationally faster code. As described by the Octave documentation [2], "Vectorization is a programming technique that uses vector operations instead of element-by-element loop-based operations. Besides frequently producing more succinct Octave code, vectorization also allows for better optimization in the subsequent implementation" In order to do this and show what differences vectorizations makes in Octave

matrix multiplication, this paper is an attempt to show our data as we use for loops to multiply our matrices and move to strict matrix-matrix multiplication enabled by Kronecker product.

This paper specifically looks at three different methods to do mapped matrix multiplication where the set is stored in a 3D matrix of matrices and a cell array of matrices. A third version is a vectorized version where a set is stored in a 2D"block" vector. Furthermore, attention is restricted to square matrices as opposed to general matrix multiplication. Solved here, given a $k \times k$ matrix A and set of n $k \times k$ matrices $\mathcal{B} = \{B_i\}$ compute the set of matrices $\mathcal{C} = \{C_i\}$ where $C_i = A * B_i$

Background

GNU Octave is a high-level interpreted language, written in C++, whose main purpose is for numerical computations. It also provides capabilities for the numerical solution of linear and nonlinear problems, but does not have data structures for multiplying sets of matrices at once as a mapped matrix multiplication. This consists of mapping, or applying, a function or operation over a set of qualified variables.

The Set as a 3D Matrix

The first way to demonstrate mapped matrix multiplication is by storing the set of matrices in a 3D matrix named B. Then by iterating through each dimension of B, one can multiply the matrix at each iteration, obtained using $B(:,:,i)$, with the original matrix, A, and store the result at the current index of B.

Algorithm 1 Mapped Matrix Multiplcation using 3D Matrix

```
function y = mappedMatMult3D(A, B)

        sizeB = size(B, 3);
        y = zeros(size(B));

        for i = 1:sizeB
                y(:,:,i) = A * B(:,:,i);
        endfor

endfunction
```

The Set as a Cell Array

The second way to demonstrate mapped multiplication is using a cell array to store the set of matrices. The Octave cell array is a container class able to store several variables of different sizes or types in one variable. Selecting and multiplying the matrices is similar to that of the 3D matrix version. The single matrix, A, will be multiplied by each cell, which is retrieved using a loop and calling B_i. Each multiplication is then stored at the index B_i, with the final output being B, a cell array.

Algorithm 2 Mapped Matrix Multiplcation using Cell Array

```
function y = mappedMatMultCell(A, B)
       sizeB = length(B);
       y = cell(1, sizeB);
  y(:) = zeros(size(A));

       for i = 1:sizeB
              y{i} = A * B{i};
       endfor

endfunction
```

The Set as a Block Matrix

The third method shows the use of vectorization in Octave, demonstrated by the use of sparse matrices and Kronecker product. B is stored as a $(k \times n) \times k$ matrix. In order to achieve mapped matrix multiplication, A is stored along the diagonal in a sparse matrix using the Kroneckor product, allowing for a single matrix multiplication with B.

Algorithm 3 Mapped Matrix Multiplcation using Vectorization

```
function y = mappedMatMultBlock(A,B)

       n = (rows(B)/columns(B));
       y = kron(speye(n),A)*B;

endfunction
```

Methods and Results

Each set of results were gathered using the same set of matrices, stored in different way according to the function. For each input, A was always the matrix formed by the command $reshape(1 : (n * n) : n, n)$. For example,

$$A = reshape(1 : (2 * 2) : 2, 2) = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

and B is always a series of $n \times n$ identity matrices stored differently according to each version. In order to obtain an $n \times n$ version of this matrix for larger and larger n, the function $eye(n)$ was used. For example,

$$B = eye(2) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

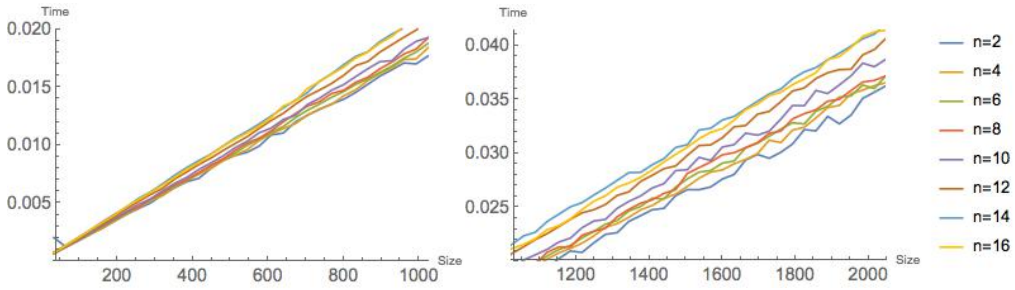
In order to gather the computation time spent on each version of the mapped matrix multiplication, Octave built-in profiling functions *tic* and *toc* were used. The function *tic* starts an internal time that is terminated by a call to *toc*. Calling *toc* also outputs the time since the timer was set. As shown in the GNU Octave documentation:

```
tic ();
# many computations later
elapsed_time = toc ();
```

Nesting a function call in between *tic* and *toc* gathered the following results.

Results For mappedMatMult3D

Shown below are the graphs for mappedMatMult3D, which uses a 3D matrix to store the set of matrices to be multiplied. The left graph represents the relation in time and the set of matrices size 32 to 1024, and the right being set sizes 1024 to 2048.

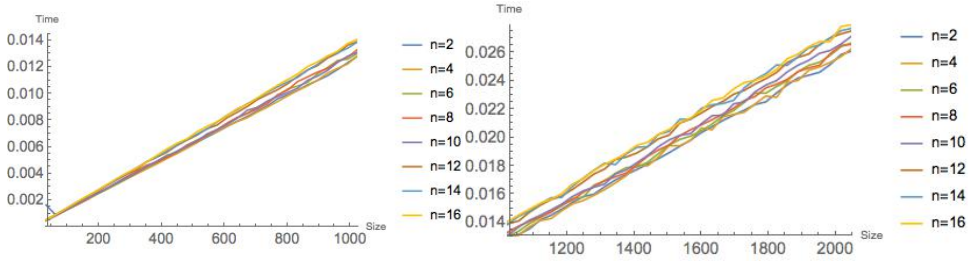


(a) Set size 32 to 1024 matrices (b) Set size 1024 to 2048 matrices

Figure 1: Time x Number of Matrices: Stored in 3D Matrix

Results For mappedMatMultCell

Shown below are the graphs for mappedMatMultCell, which uses a cell array to store the set of matrices to be multiplied. The left graph represents the relation in time and the set of matrices size 32 to 1024, and the right being set sizes 1024 to 2048.

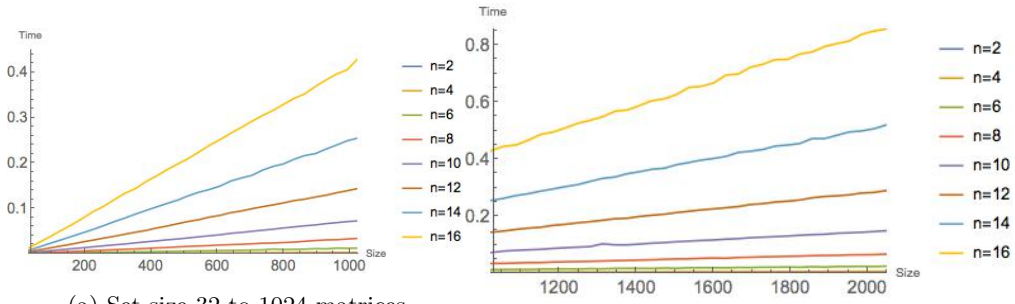


(a) Set size 32 to 1024 matrices (b) Set size 1024 to 2048 matrices

Figure 2: Time x Number of Matrices: Stored in Cell Array

Results For mappedMatMultBlock

Shown below are the graphs for mappedMatMultBlock, which uses a block matrix to store the set of matrices to be multiplied. The left graph represents the relation in time and the set of matrices size 32 to 1024, and the right being set sizes 1024 to 2048.



(a) Set size 32 to 1024 matrices

(b) Set size 1024 to 2048 matrices

Figure 3: Time x Number of Matrices: Stored in 2D Block Matrix

Conclusions and Future Work

As shown in Figure 1, as the size of the $k \times k$ matrices increase, the difference in the amount of computation time grows slowly. In Figure 2, storing the matrices in a cell array causes a very tight group of computation times, varying very little as k grows. Finally, in Figure 3, it can be seen that storing the matrices in a block matrix and using the Kronecker product has a significantly small computation time with small $k \times k$, but drastically increases as k grows in size.

The graphs below read left to right, showing the progression of each algorithm as the size of the matrices in each set grow. The blue line represents the first algorithm, mappedMatMult3d. The orange representing the second algorithm, mappedMatMultCell. The green representing the last algorithm, mappedMultMatBlock. The x-axis shows the set size of the matrices, and the y-axis shows the time in seconds for the computation to run.

As shown below, mappedMatMultBlock is computationally faster than the others for matrices up to size 6×6. It is also important to note that they mappedMatMultCell is consistently faster than the other algorithms as the size of the matrices grow, so for moderately sized matrices using a cell array seems better.

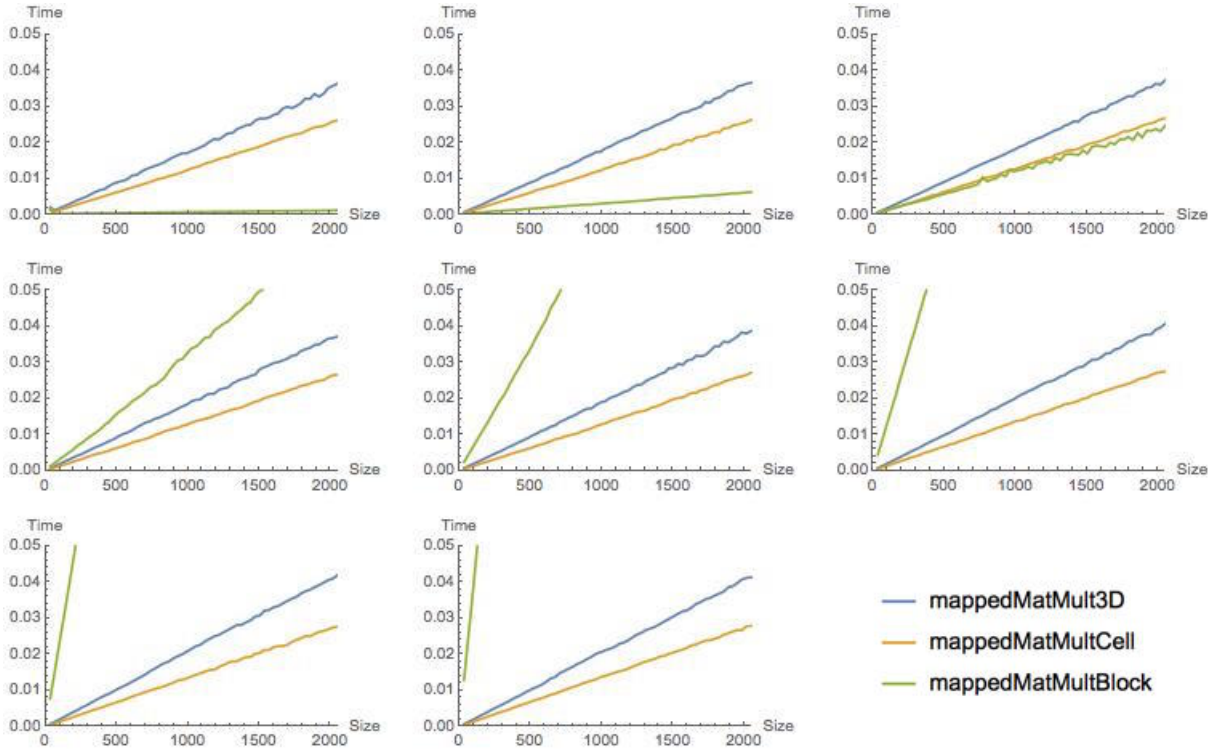


Figure 4: Progression of all 3 versions for different matrix versions

One area of future research that exists is using GPU-accelerated computing with CUDA drop-in acceleration libraries. This will enable use of the computers GPU in order to speed up computation time with massively parallel power. [3] We would also like to explore the idea of storing not only the square matrices, but allowing for matrices of any dimension.

References

[1] Gnu octave. https://www.gnu.org/software/octave/, 2014.

[2] John W. Eaton, David Bateman, Soren Hauberg, and Rik Wehbring. *GNU Octave: Free Your Numbers*, 2011.

[3] Nikolay Markovskiy. Drop-inacceleration of gnu octave. http://devblogs.nvidia.com/parallelforall/drop-in-acceleration-gnu-octave/, 2014.