Blue Waters Petascale Semester Curriculum v1.0

Unit 9: Optimization

Lesson 1: Cache Efficient Matrix Multiplication

Instructor Guide

Developed by Paul F. Hemler for the Shodor Education Foundation, Inc.

Except where otherwise noted, this work by The Shodor Education Foundation, Inc. is licensed under CC BY-NC 4.0. To view a copy of this license, visit
https://creativecommons.org/licenses/by-nc/4.0

Browse and search the full curriculum at
http://shodor.org/petascale/materials/semester-curriculum

We welcome your improvements! You can submit your proposed changes to this material and the rest of the curriculum in our GitHub repository at
https://github.com/shodor-education/petascale-semester-curriculum

We want to hear from you! Please let us know your experiences using this material by sending email to petascale@shodor.org

Effective Caching for Matrix Multiplication

Paul F. Hemler, Hampden-Sydney College

Abstract

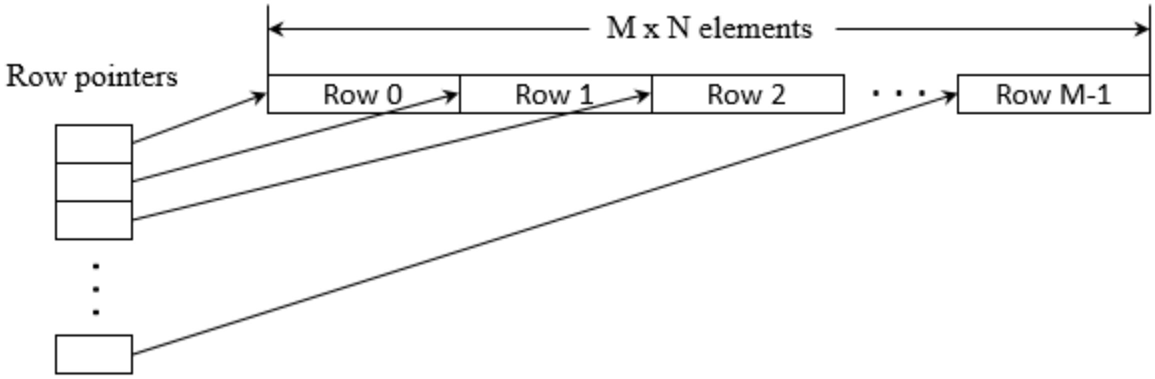
In this module the student will experiment with a program and a Bash shell script to empirically show that program execution time is dependent on accessing the elements in a cache friendly manner when the application is matrix multiplication. The module then demonstrates how the cache can be efficiently accessed in a two-dimensional array and how matrix multiply can be made more cache friendly by first transposing one of the matrices. Timing results for a variety of square matrices are then plotted clearly showing that for large matrices multiplying a matrix by the transpose of another matrix better utilizes the cache, resulting in faster program execution.

Upon completion of this module the student should understand the standard matrix multiplication algorithm utilizing a two-dimensional array and understand why this method does not effectively utilize the cache memory. Finally, the student should lean that more efficient matrix multiplication can be implemented by first transposing the elements of one of the matrices before multiplication.

Introduction

A two-dimensional array is a very common way of storing, maintaining, and processing data in a variety of scientific domains. For example, a two-dimensional array can represents a Matrix in the field of Linear Algebra or an image or two-dimensional spatial domain representation in Engineering. In this experiment students will demonstrate that accessing the data in a two-dimensional array is fundamentally important for efficient program execution when performing matrix multiplication.

Even though a computer's memory is physically organized as a linear (one-dimensional) array it is a simple matter to provide the programmer a conventional two-dimensional logical organization. This capability is supplied by allocating a linear array of pointers, where the pointers will be resolved to the first element in each row, as shown in the figure below.



Utilizing the row pointers allows the programmer to access array elements in the more natural way as MatrixName[row index][column index] instead of addressing the elements in the linear array as MatrixName[row index * number of columns + column index.

When the rows are stored in sequential memory locations (as the figure above shows) it is referred to as row-major order, which is the standard for the **C** language. The FORTRAN language stores the columns in sequential memory locations, referred to as column-major order. There is not a technical advantage for either approach but as will be demonstrated below there is an advantage to using both orderings for matrix multiplication as the matrices get large.

Quick review questions

1. What would need to change to allocate and access the elements of a three-dimensional array?

Matrix Multiplication

Matrices are found in many different Science and Engineering problems. A matrix is simply a two-dimensional array of numbers.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

To multiply two matrices together, the number of columns of the matrix on the left must be the same as the number of rows of the matrix on the right. In this module we will just multiply square matrices, which means all matrices have the same number of rows as columns. The final result of multiplying two matrices together is a third matrix of the same size when multiplying square matrices. The process can be described by the equation:

$$\boldsymbol{C} = \boldsymbol{AB},$$

where ***C*** is also a square matrix with the same number of rows and columns as matrices ***A*** and ***B***. The *(i, j)* element in the resulting matrix ***C*** is determined by forming the sum of the element by element multiplication the ith row of matrix ***A*** by the jth column of matrix ***B***. Formally this can be written mathematically as:

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

This equation shows that each element in the product matrix requires *n* multiplications and *n-1* additions. Since there are n^2 computed elements in the resulting array (***C***), we say matrix multiplication is an n^3 operation, where *n* is the size of the matrix.

Quick review questions

2. How would this change to allow a non-square matrix multiplication?
3. Write a function that computes the sum of the product of all the elements of the i^{th} row with the j^{th} column.

Program description

The program is called ***matMult.c***. The program requires the user to supply an integer parameter specifying the size of the two square matrices that are to be multiplied together. The four two-dimensional arrays to represent the different matrices are then allocated by first allocating the row pointers followed by allocating memory space for the elements in the array. Finally, the row pointers are resolved to contain the address of each row.

```
double** mat1 = (double**)malloc(nRows * sizeof(double*));
mat1[0] = (double*)malloc(nRows * nCols * sizeof(double));
for (r = 1; r < nRows; ++r)
      mat1[r] = &(mat1[0][r * nCols]);
```

The program uses four matrices so that the product matrix can be computed two different ways and the result can be checked for correctness.

Once the matrices are allocated and configured two of the matrices are populated with known values. The first matrix mat1 is:

$$mat1 = \begin{bmatrix} 1 & 2 & \cdots & n \\ n+1 & n+2 & \cdots & 2n \\ \vdots & \vdots & \ddots & \vdots \\ (n-1)n+1 & (n-1)n+2 & \cdots & n^2 \end{bmatrix}$$

And the second matrix is:

$$mat2 = \begin{bmatrix} n^2 & n^2-1 & \cdots & (n-1)n+1 \\ (n-1)n & (n-1)n-1 & \cdots & (n-1)n-(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ n & n-1 & \cdots & 1 \end{bmatrix}$$

Once *mat1* and *mat2* have been initialized the standard matrix multiply is performed and timed:

```
for (int r = 0; r < nRows; ++r)
      for (int c = 0; c < nCols; ++c) {
            mat3[r][c] = 0.0;
            for (int i = 0; i < nRows; ++i)
                  mat3[r][c] += mat1[r][i] * mat2[i][c];
      }
```

The transpose of a matrix is a simple reordering so that the rows become the columns. This operation is performed on *mat2* and timed using the following code:

```
for (int r = 0; r < nRows; ++r)
      for (int c = r + 1; c < nCols; ++c) {
            val = mat2[r][c];
            mat2[r][c] = mat2[c][r];
            mat2[c][r] = val;
      }
```

Finally, the matrix product is computed by multiplying the i^{th} row of mat1 with the j^{th} ***row*** of *mat2.*

The time for the standard matrix multiply, the matrix transpose, and the transpose multiply are then displayed. The last processing step in the program checks that both matrix multiplies results in the same matrix. This is done by summing the squared element by element difference between the two matrices and testing for a non-zero value.

Block Matrix Multiply

One of the problems with performing matrix multiply is that the elements are used multiple times to compute various elements in the product matrix. For example, row zero is multiplied by all the columns starting with the first column and ending with the last column. Once the first row of the product matrix has been computed the process repeats with row one, which is multiplied by all the columns. When the matrices are large compared to the size of the cache, it is very unlikely the column data still resided in the cache and it must be retrieved from main memory. One way to better utilize the cache is to perform all required computations when the data is in the cache. This avoids the cache misses that increases the run time.

Block matrix multiply is a technique that capitalizes on this idea. A matrix can be thought of as comprised of square blocks, where each block is a sub-matrix of the original matrix. For example,

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} & \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix} \\ \begin{bmatrix} a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{bmatrix} & \begin{bmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix}$$

Given a similar block partitioning for the B and C, the product matrix can be computed as:

$$C = \begin{bmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix}$$

The product matrix is:

$$\begin{aligned} C_{0,0} &= A_{0,0}B_{0,0} + A_{0,1}B_{1,0} \\ C_{0,1} &= A_{0,0}B_{0,1} + A_{0,1}B_{1,1} \\ C_{1,0} &= A_{1,0}B_{0,0} + A_{1,1}B_{1,0} \\ C_{1,1} &= A_{1,0}B_{0,1} + A_{1,1}B_{1,1} \end{aligned}$$

Each of the C's is a 2 X 2 matrix and the idea of block matrix multiply is to compute each block while the corresponding blocks are in the cache. Of course, the idea scales up for larges size matrices and blocks.

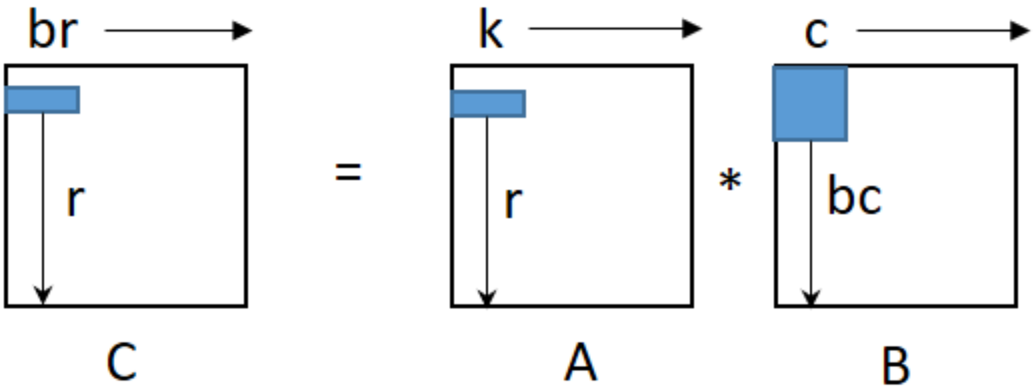
Program description

The function to perform block matrix multiplication is a bit more complicated compared to serial matrix multiply because there is more book keeping to multiply the blocks.

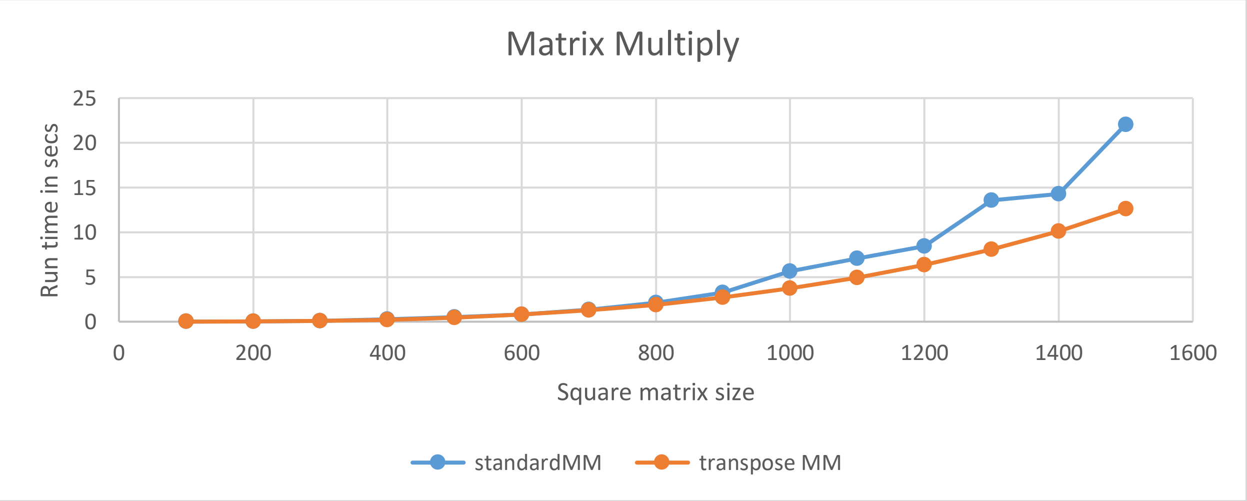
```
// For all row blocks
for (br = 0; br < n; br += blockSize) {
      // For all column blocks
      for (bc = 0; bc < n; bc += blockSize) {
            // Fill in a sliver of C
            for (r = 0; r < n; ++r) {
                  // Multiply the 1 X blockSize sliver (partial row) of
                  // A with the blockSize X blockSize block of B and store
                  // the results in the 1 X blockSize sliver of C
                  for (c = br; c < myMin(br + blockSize, n); ++c) {
                        sum = 0.0;
                        for (k = bc; k < myMin(bc + blockSize, n); ++k)
                              sum += A[r][k] * B[k][c];
```

```
                    C[r][c] += sum;
                }
            }
        }
}
```

Essentially, the function computes a 1 X blockSize slice of the *C* matrix for all of the rows as shown in the figure below.

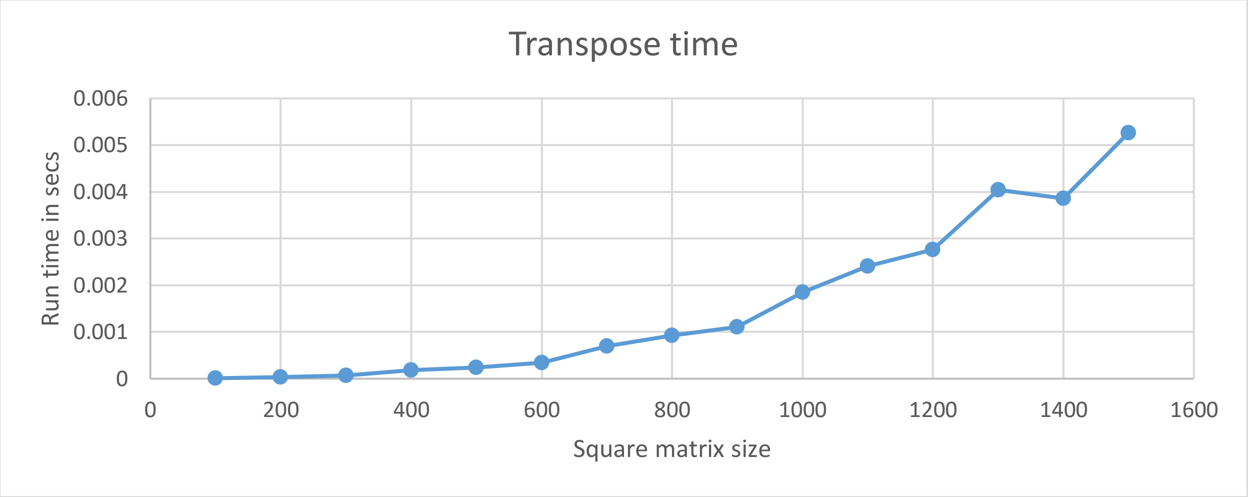


Results
A bash script was written ***runMatMult.sh*** to execute the ***matMult.c*** program with square matrices of size, 100, 200, 300, …, 1500. Double-precision floating-point numbers were used for the matrices. The running time for both the standard and transpose matrix multiplication are shown in the figure below.



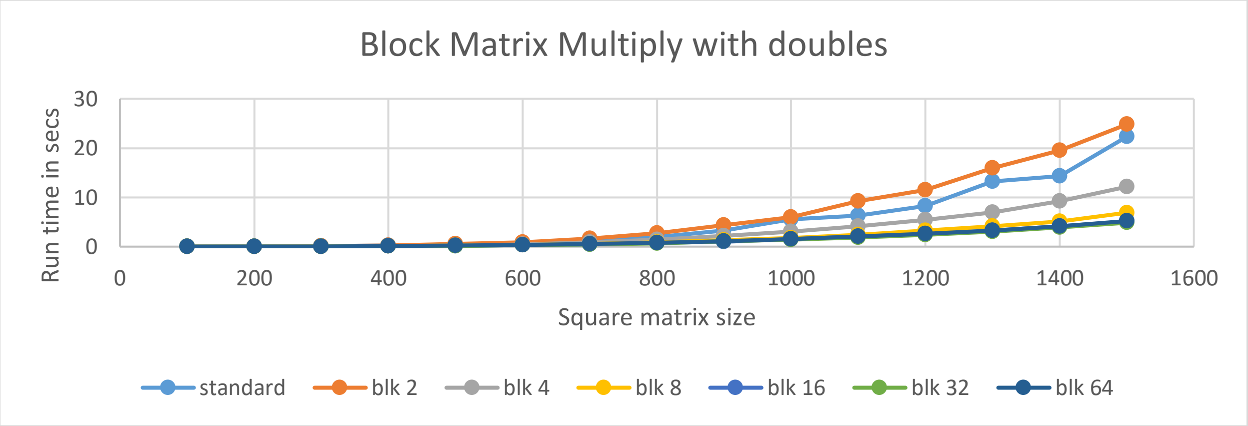
The saving in matrix multiplication time begins when the size of the matrices are 900 x 900 elements and then the time saving is more significant as the matrices get larger.

It is surprising that the time it takes to perform the transpose operation is relatively constant and is shown in the figure below.

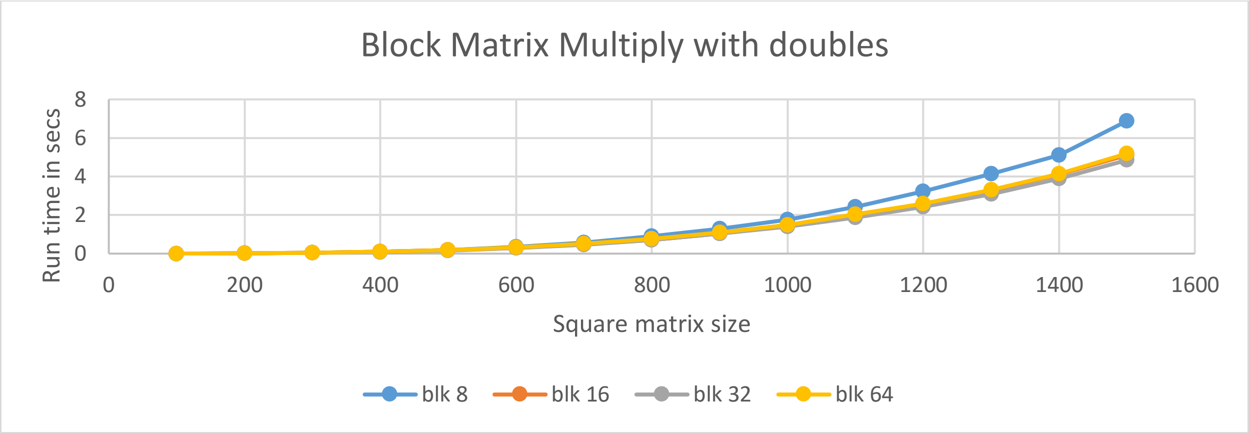


This experimentation suggests that matrix multiplication performed with row data is more efficient for matrices of size 1000 X 1000 and larger. For smaller matrices the performance gain of a more cache efficient algorithm is not significant.

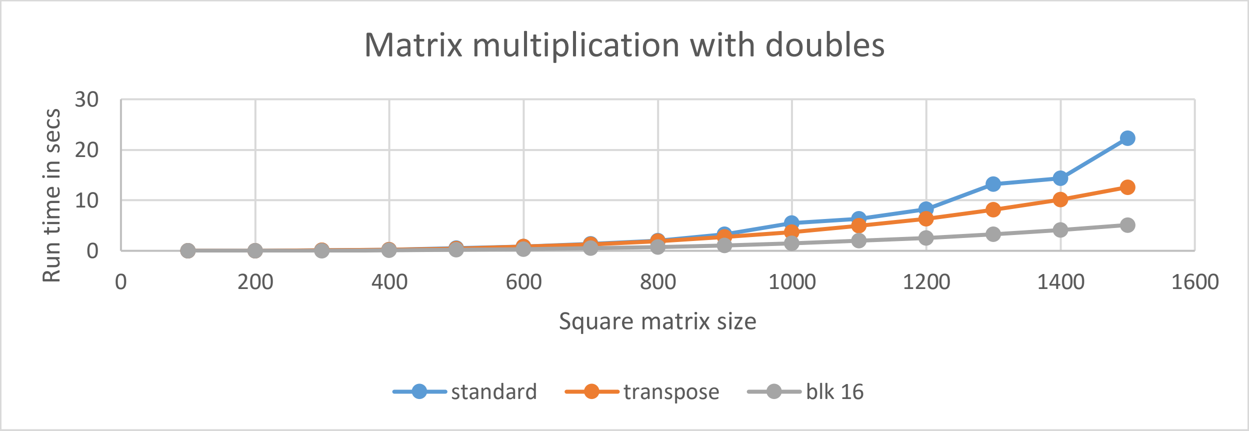
The block matrix multiply program allows the block size to be controlled on the command line and its values were 2, 4, 8, 16, 32, and 64. The figure below shows the running times as a function of block and matrix sizes.



When the block size was set to 2, block matrix multiplication took longer than simple matrix multiplication. This is because of extra computation and control for the additional loops. Apparently, there was not any saving in cache hit reduction to make up for the extra execution time. A block size of 4 resulted in a better performing program for block matrix multiplication and additional improvement is observed when the block size was 8. There was a slight improvement in performance when increasing the block size but setting it at 8 or 16 balances the extra control time with reduced cache misses. A figure showing only block sizes of 8 and larger is shown below.



One final figure shown below compares the run times of standard, transpose, and block matrix multiply for the various size matrices used in this paper. Clearly, the performance of block matrix multiply is superior when for matrices of 1000 X 1000 and large. The reason for the better performance is because there are fewer cache misses so the computation and not the memory bottle neck is the limiting factor.


Matrix multiplication with doubles
Run time in secs
Square matrix size
standard
transpose
blk 16