

# OpenCL Matrix Multiplication



**Oregon State**  
University  
**Mike Bailey**

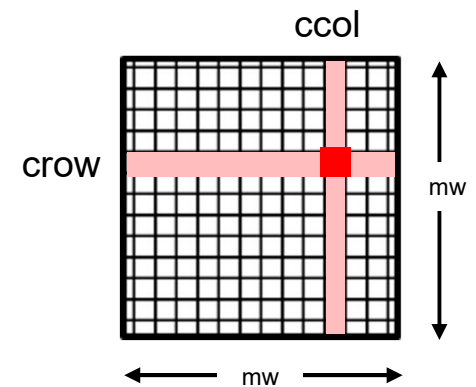
mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



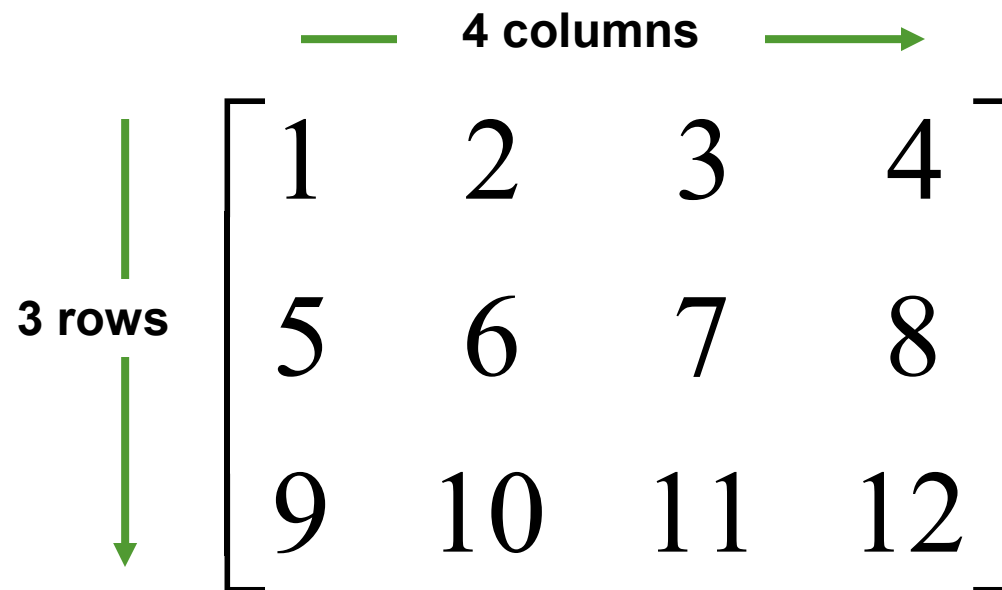
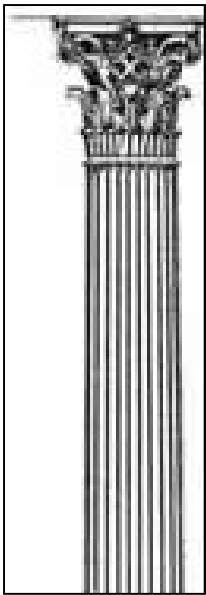
**Oregon State**  
University  
Computer Graphics



## Matrices

**A matrix is a 2D array of numbers, arranged in rows that go across and columns that go down:**

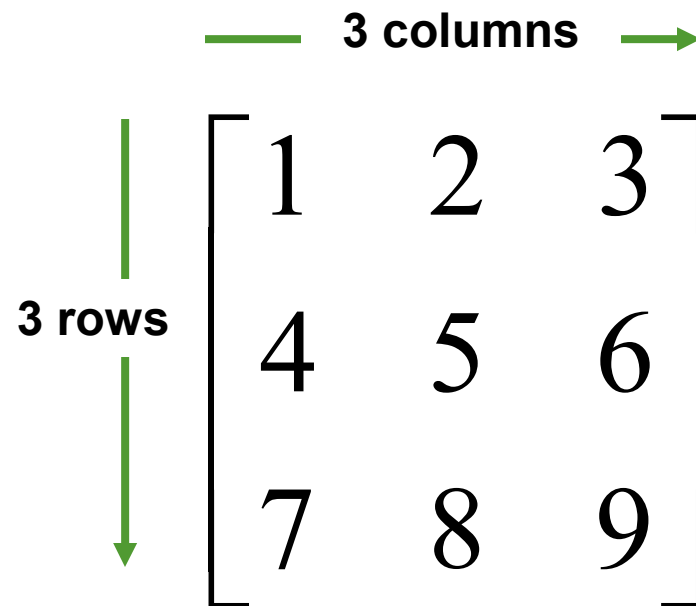
A column:



Matrix sizes are termed “#rows x #columns”, so this is a 3x4 matrix

## Square Matrices

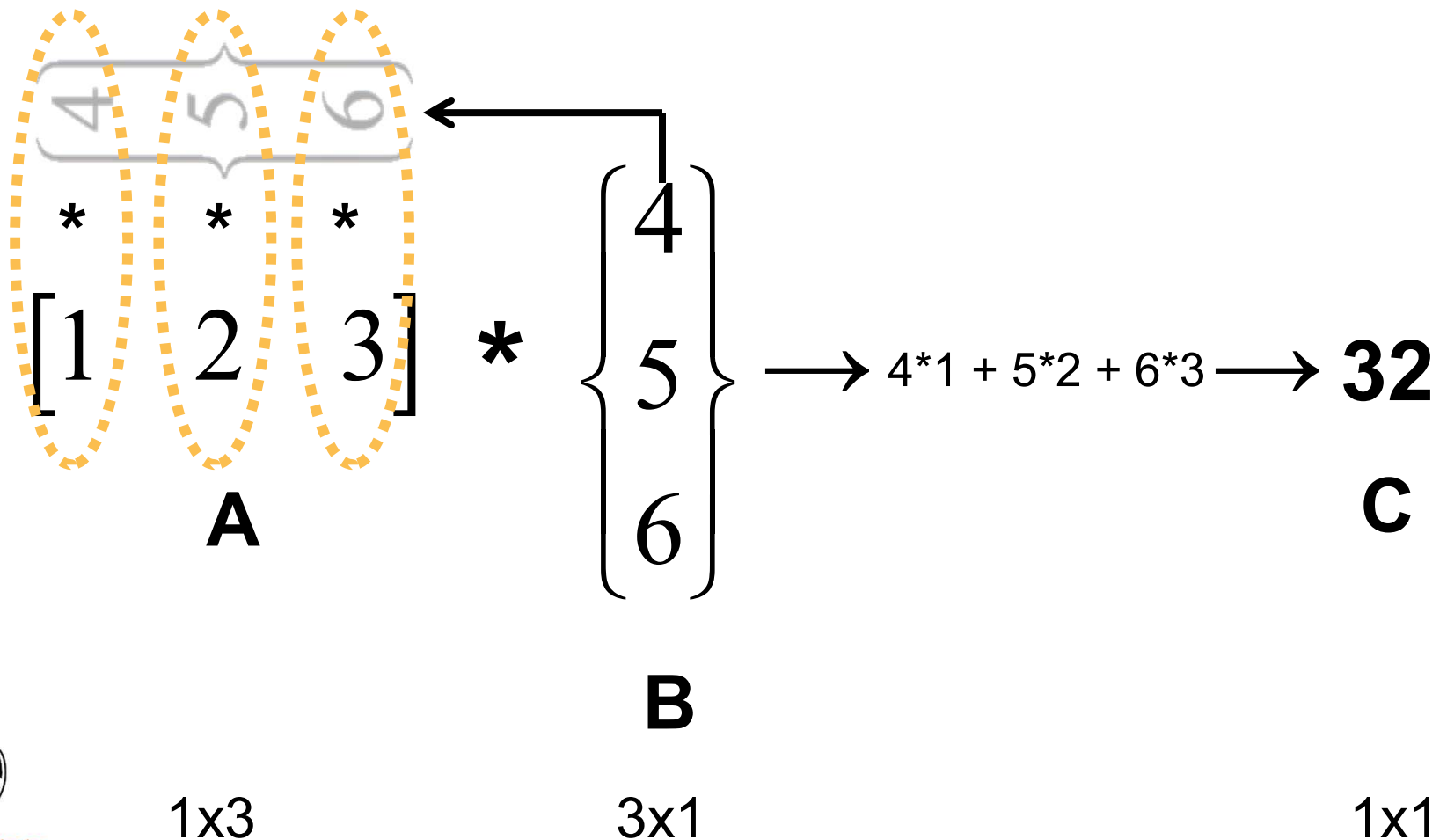
A square matrix has the same number of rows and columns



This is a 3x3 matrix

## Matrix Multiplication

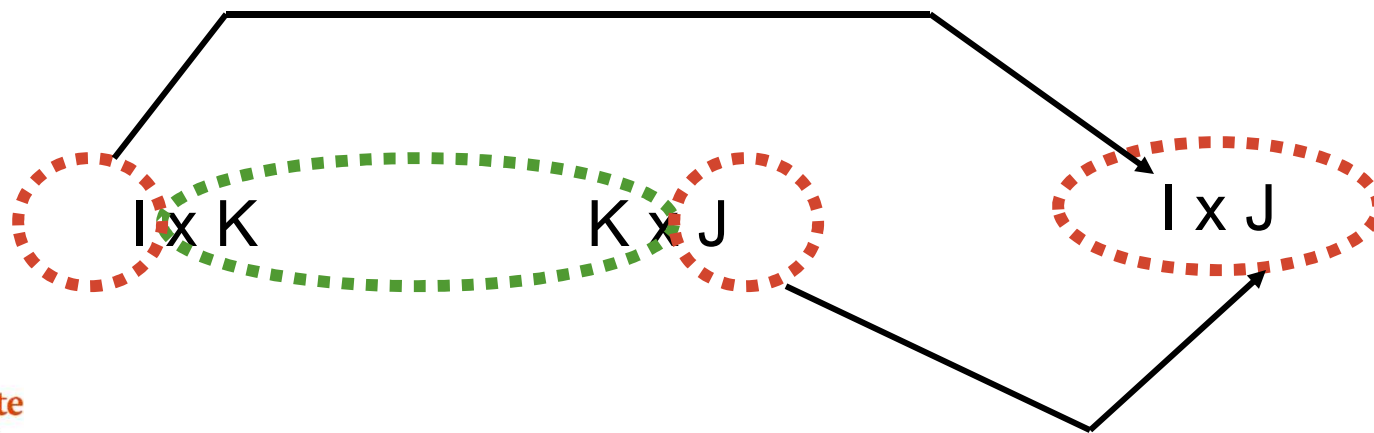
The basic operation of matrix multiplication is to pair-wise multiply a single row by a single column



## Matrix Multiplication

Two matrices, A and B, can be multiplied if the number of columns in A equals the number of rows in B. The result is a matrix that has the same number of rows as A and the same number of columns as B.

$$\begin{bmatrix} [1 & 2 & 3] \\ \mathbf{A} \end{bmatrix} * \begin{bmatrix} \begin{Bmatrix} 4 \\ 5 \\ 6 \end{Bmatrix} \\ \mathbf{B} \end{bmatrix} \Rightarrow \begin{bmatrix} 32 \\ \mathbf{C} \end{bmatrix}$$



## Matrix Multiplication in Software

Here's how to remember how to do it:

1.  $C = A * B$

$$\begin{bmatrix} 1 & 2 & 3 \\ \mathbf{A} \end{bmatrix} * \begin{bmatrix} \begin{Bmatrix} 4 \\ 5 \\ 6 \end{Bmatrix} \\ \mathbf{B} \end{bmatrix} \Rightarrow \begin{bmatrix} 32 \\ \mathbf{C} \end{bmatrix}$$

2.  $[I \times J] = [I \times K] * [K \times J]$

$$I \times J = I \times K \times K \times J$$

The diagram shows the dimensions of the matrices in the second equation. The first term,  $I \times J$ , is enclosed in a red dashed oval. The second term,  $I \times K$ , is enclosed in a red dashed oval. The third term,  $K \times J$ , is enclosed in a red dashed oval. A green dashed oval encloses the entire product  $I \times K \times K \times J$ , indicating that the dimensions of the result match the dimensions of the first term.

$$C[i][j] += A[i][k] * B[k][j];$$

## Matrix Multiplication in CPU Software

$$\begin{bmatrix} 1 & 2 & 3 \\ \mathbf{A} \end{bmatrix} * \begin{bmatrix} 4 \\ 5 \\ 6 \\ \mathbf{B} \end{bmatrix} \Rightarrow \begin{bmatrix} 32 \\ \mathbf{C} \end{bmatrix}$$

```

for( int i = 0; i < numRows; i++ )
{
    for( int j = 0; j < numBcols; j++ )
    {
        C[ i ][ j ] = 0.;
        for( int k = 0; k < numAcols; k++ )
        {
            C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
        }
    }
}

```

## Matrix Multiplication in CPU Software

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \underset{\mathbf{A}}{*} \underset{\mathbf{B}}{\begin{Bmatrix} 4 \\ 5 \\ 6 \end{Bmatrix}} \Rightarrow \underset{\mathbf{C}}{\begin{bmatrix} 32 \end{bmatrix}}$$

Note that saying:

```
C[ i ][ j ] = 0.;
for( int k = 0; k < numAcols; k++ )
{
    C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
}
```

Is like saying:

**C[ i ][ j ] = A[i][0] \* B[0][j] + A[i][1] \* B[1][j] + A[i][2] \* B[2][j] + A[i][3] \* B[3][j] ...**





```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>
#include "cl.h"
#include "cl_platform.h"

// the matrix-width and the number of work-items per work-group:
// note: the matrices are actually MATWxMATW and the work group sizes are LOCALSIZExLOCALSIZE:
#ifndef MATW
#define MATW      1024
#endif

#ifndef LOCALSIZE
#define LOCALSIZE  8
#endif

// opengl objects:
cl_platform_id      Platform;
cl_device_id        Device;
cl_kernel           Kernel;
cl_program          Program;
cl_context           Context;
cl_command_queue    CmdQueue;

float               hA[MATW][MATW];
float               hB[MATW][MATW];
float               hC[MATW][MATW];

const char *        CL_FILE_NAME = { "proj06.cl" };
```

# The .cl Kernel Function

10

```
#define IN
#define OUT
```

```
kernel
void
```

```
MatrixMult( IN global const float *dA, IN global const float *dB, IN global int *dMW, OUT global float *dC )
```

```
{
```

```
    // [dA] is dMW x dMW
```

```
    // [dB] is dMW x dMW
```

```
    // [dC] is dMW x dMW
```

```
    // but all the matrixs' rows are really linear in memory
```

```
    int mw = *dMW;
```

```
    int crow = get_global_id( 0 );    // which row we are filling
```

```
    int ccol = get_global_id( 1 );    // which column we are filling
```

```
    int aindex = crow * mw;           // a[i][0]
```

```
    int bindex = ccol;                 // b[0][j]
```

```
    int cindex = crow * mw + ccol;     // c[i][j]
```

```
    float cij = 0.;
```

```
    for( int k = 0; k < mw; k++ )
```

```
    {
```

```
        cij += dA[aindex] * dB[bindex];
```

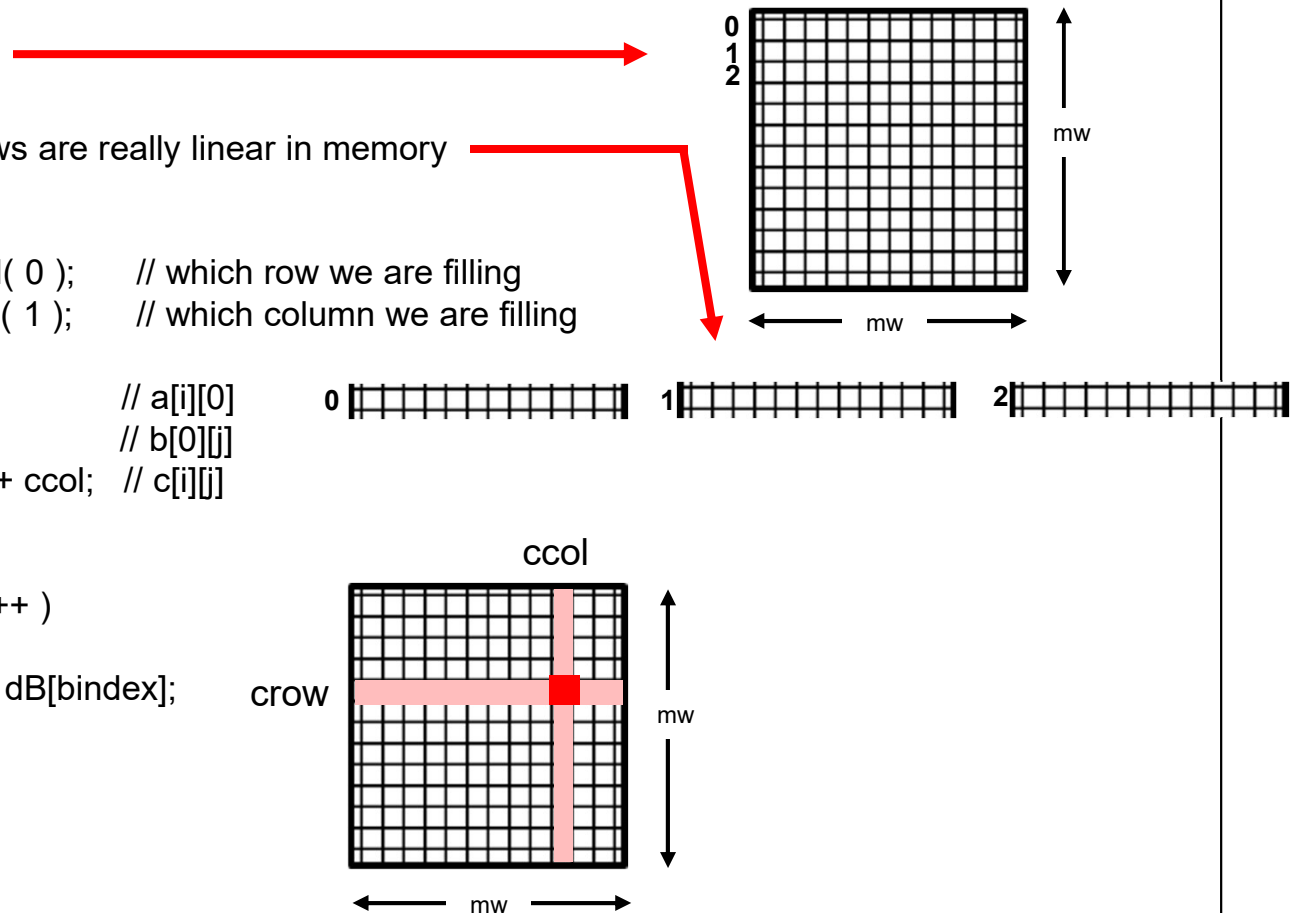
```
        aindex++;
```

```
        bindex += mw;
```

```
    }
```

```
    dC[cindex] = cij;
```

```
}
```



## Setting Up the Memory for the Matrices

11

```
int mw = MATW;  
size_t aSize = MATW * MATW * sizeof(float);  
size_t bSize = MATW * MATW * sizeof(float);  
size_t mwSize = sizeof(mw);  
size_t cSize = MATW * MATW * sizeof(float);  
  
cl_mem dA = clCreateBuffer( Context, CL_MEM_READ_ONLY, aSize, NULL, &status );  
...  
  
status = clEnqueueWriteBuffer( CmdQueue, dA, CL_FALSE, 0, aSize, hA, 0, NULL, NULL );  
...  
  
Wait( CmdQueue );
```

Remember that our kernel function looks like this:

```
kernel  
void  
MatrixMult( IN global const float *dA, IN global const float *dB, IN global int *dMW, OUT global float *dC )
```

So the definition of the arguments needs to look like this:

```
Kernel = clCreateKernel( Program, "MatrixMult", &status );  
  
status = clSetKernelArg( Kernel, 0, sizeof(cl_mem), &dA );  
status = clSetKernelArg( Kernel, 1, sizeof(cl_mem), &dB );  
status = clSetKernelArg( Kernel, 2, sizeof(cl_mem), &dMW );  
status = clSetKernelArg( Kernel, 3, sizeof(cl_mem), &dC );
```

```
size_t globalWorkSize[3] = { MATW,      MATW,      1 };
size_t localWorkSize[3]  = { LOCALSIZE, LOCALSIZE, 1 };

Wait( CmdQueue );

double time0 = omp_get_wtime( );

status = clEnqueueNDRangeKernel( CmdQueue, Kernel, 1, NULL,
                                   globalWorkSize, localWorkSize, 0, NULL, NULL );

Wait( CmdQueue );
double time1 = omp_get_wtime( );
```

```
// performance in giga-multiplies performed per second:  
  
fprintf( stderr, "GigaMultsPerSecond: %10.2lf\n",  
         (double)MATW*(double)MATW*(double)MATW/(time1-time0)/1000000000. );
```

## Copying the Resulting Matrix from the Device back to the Host

15

```
status = clEnqueueReadBuffer( CmdQueue, dC, CL_FALSE, 0, cSize, hC, 0, NULL, NULL );  
Wait( CmdQueue );
```

## Cleaning Up

16

```
clReleaseKernel(          Kernel );  
clReleaseProgram(        Program );  
clReleaseCommandQueue( CmdQueue );  
  
clReleaseMemObject(      dA );  
clReleaseMemObject(      dB );  
clReleaseMemObject(      dMW );  
clReleaseMemObject(      dC );
```