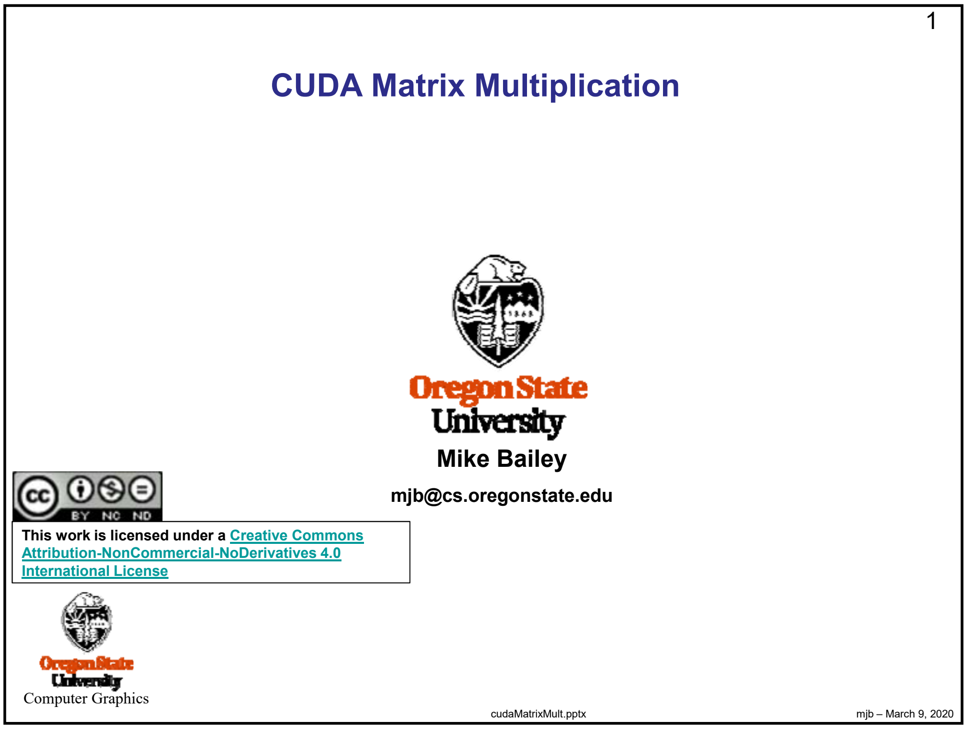
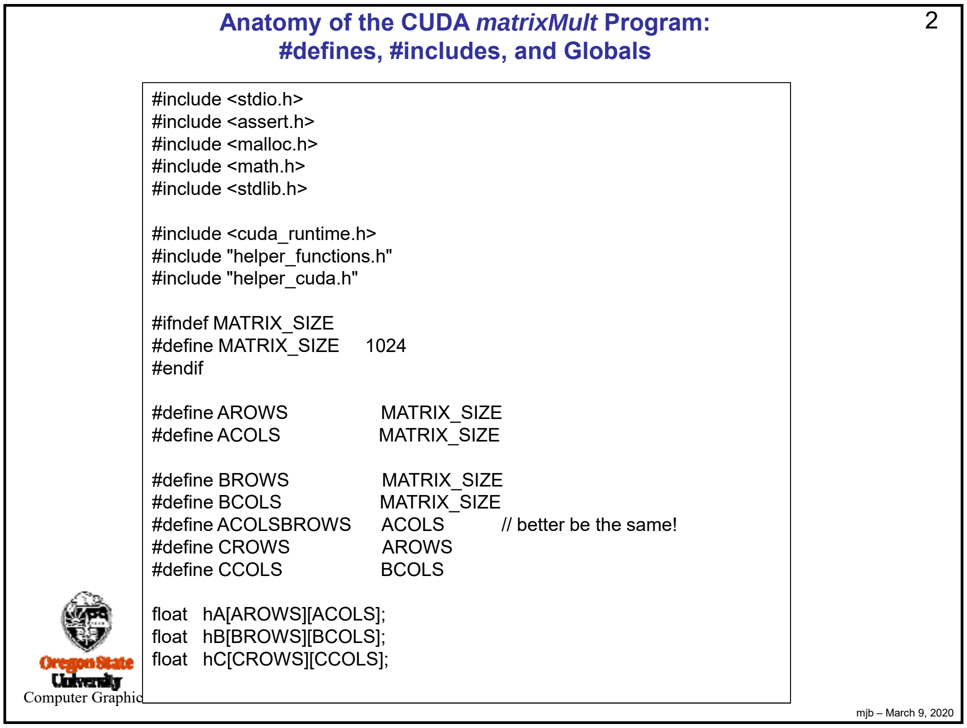


CUDA Matrix Multiplication

Mike Bailey

mjb@cs.oregonstate.edu



Anatomy of the CUDA *matrixMult* Program: #defines, #includes, and Globals

```
#include <stdio.h>
#include <assert.h>
#include <malloc.h>
#include <math.h>
#include <stdlib.h>

#include <cuda_runtime.h>
#include "helper_functions.h"
#include "helper_cuda.h"

#ifndef MATRIX_SIZE
#define MATRIX_SIZE     1024
#endif

#define AROWS           MATRIX_SIZE
#define ACOLS           MATRIX_SIZE

#define BROWS           MATRIX_SIZE
#define BCOLS           MATRIX_SIZE
#define ACOLSBROWS      ACOLS       // better be the same!
#define CROWS           AROWS
#define CCOLS           BCOLS

float   hA[AROWS][ACOLS];
float   hB[BROWS][BCOLS];
float   hC[CROWS][CCOLS];
```

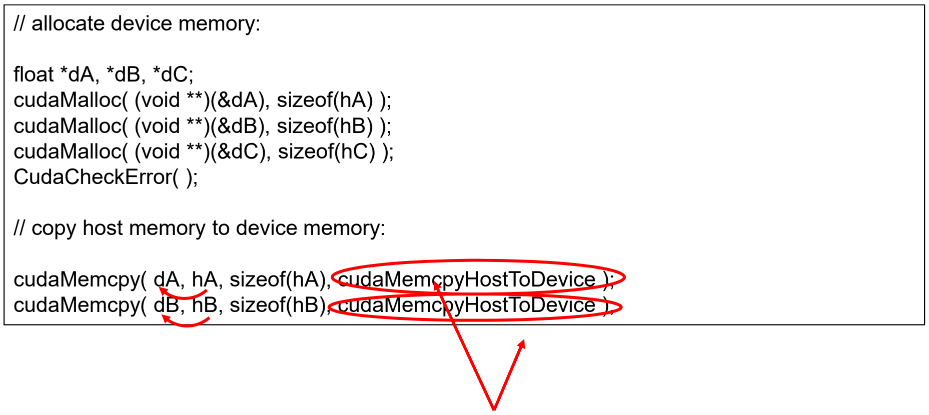
Anatomy of a CUDA Program:
Error-Checking

```
void
CudaCheckError( )
{
	cudaError_t e = cudaGetLastError( );
	if( e != cudaSuccess )
	{
		fprintf( stderr, "CUDA failure %s:%d: '%s'\n", __FILE__, __LINE__, cudaGetErrorString(e));
	}
}
```

Anatomy of a CUDA Program:
The Kernel Function

```
__global__ void MatrixMul( float *A, float *B, float *C )
{
	// [A] is AROWS x ACOLS
	// [B] is BROWS x BCOLS
	// [C] is CROWS x CCOLS  =  AROWS x BCOLS

	int blockNum      = blockIdx.y*gridDim.x + blockIdx.x;
	int blockThreads = blockNum*blockDim.x*blockDim.y;
	int gid                  = blockThreads + threadIdx.y*blockDim.x + threadIdx.x;

	int crow = gid  /  CCOLS;
	int ccol  = gid % CCOLS;

	int aindex = crow * ACOLS;                  // a[i][0]
	int bindex = ccol;                                // b[0][j]
	int cindex = crow * CCOLS + ccol;       // c[i][j]

	float cij = 0.;
	for( int k = 0; k < ACOLSBROWS; k++ )
	{
		cij += A[aindex] * B[bindex];
		aindex++;
		bindex += BCOLS;
	}
	C[cindex] = cij;
	__syncthreads( );
}
```

Anatomy of a CUDA Program:
Setting Up the Memory for the Matrices



```
// allocate device memory:

float *dA, *dB, *dC;
cudaMalloc( (void **)(&dA), sizeof(hA) );
cudaMalloc( (void **)(&dB), sizeof(hB) );
cudaMalloc( (void **)(&dC), sizeof(hC) );
CudaCheckError( );

// copy host memory to device memory:

cudaMemcpy( dA, hA, sizeof(hA), cudaMemcpyHostToDevice );
cudaMemcpy( dB, hB, sizeof(hB), cudaMemcpyHostToDevice );
```

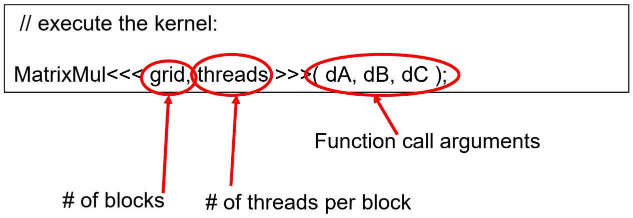
This is a defined constant in one of the CUDA .h files

In **cudaMemcpy( ),** it's always the second argument getting copied to the first!

Anatomy of a CUDA Program:
Getting Ready to Execute

```
 // setup execution parameters:
dim3 threads( 16, 16, 1 );
if( threads.x > CROWS )
                threads.x = CROWS;
if( threads.y > CCOLS )
                threads.y = CCOLS;
dim3 grid( CROWS / threads.x, CCOLS / threads.y );

// create cuda events for timing:
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop  );
CudaCheckError( );

// record the start event:
cudaEventRecord( start, NULL );
```

Anatomy of a CUDA Program: Executing the Kernel



```
// execute the kernel:

MatrixMul<<< grid, threads >>>( dA, dB, dC );
```

Function call arguments

# of blocks

# of threads per block

The call to **MatrixMul**( ) returns *immediately*!

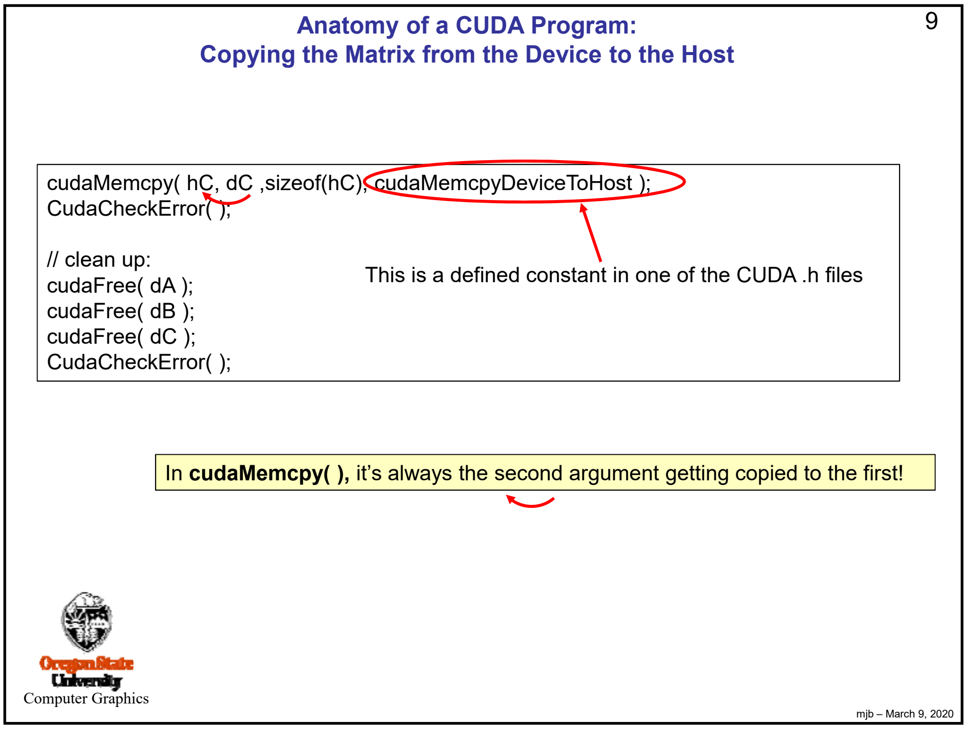
If you upload the resulting array (dC) right away, it will have garbage in it.

To block until the kernel is finished, call:

cudaDeviceSynchronize( );

Anatomy of a CUDA Program: Getting the Stop Time and Printing Performance

```
// record the stop event:
cudaEventRecord( stop, NULL );

// wait for the stop event to complete:
cudaEventSynchronize( stop );

float msecTotal;
cudaEventElapsedTime( &msecTotal, start, stop );

// performance:
float msecPerMatrixMul = msecTotal;
double flopsPerMatrixMul = (double)CROWS * (double)CCOLS * (double)ACOLSBROWS;
double gigaFlops = ( flopsPerMatrixMul / 1000000000. ) / ( msecPerMatrixMul / 1000.0 ) ;
fprintf( stderr, "%6d\t%6d\t%10.3lf\n", CROWS, CCOLS, gigaFlops );
```



Anatomy of a CUDA Program:
Copying the Matrix from the Device to the Host

```
cudaMemcpy( hC, dC ,sizeof(hC), cudaMemcpyDeviceToHost );
CudaCheckError( );

// clean up:
cudaFree( dA );
cudaFree( dB );
cudaFree( dC );
CudaCheckError( );
```

This is a defined constant in one of the CUDA .h files

In **cudaMemcpy( ),** it's always the second argument getting copied to the first!

Oregon State University
Computer Graphics