```
/*
  CSci 360 Computer Architecture 3
  Hunter College of the City University of New York
  Prof. Stewart Weiss

  CUDA-based MATRIX MULTIPLICATION Without Using Shared Memory

  This program demonstrates a common CUDA paradigm, as well as how CUDA
  relates to C/C++.  It is not a main program, but a function that can
  utilize the resources of the GPU to perform matrix multiplication.
  It computes the matrix product C = A*B, where A and B are matrices of
  type float.

  BACKGROUND

  CUDA extends the set of types of C/C++ with types whose names are of the form
      type1, type2, type3, and in some cases, type4

  where "type" can be replaced by one of:
      char, uchar, int, uint, long, ulong, short, ushort, or float

  For example, char3, uint4, and ulong3 are all type names in CUDA.
  These are called "vectors" in CUDA, but they are really C structs with members
  x, y, z, and w respectively.  Thus, for example, if Point is declared as

      int3 Point;

  then Point.x, Point.y, and Point.z are the members of Point. These types all
  have constructors of the form make_type(arg1,...,arg4). Thus we can initialize
  a member as follows:

      uint3 Dimensions_of_box(100,200,200);

  The type dim3 is a an extension of uint3. It specifies dimensions of things,
  but the constructor initializes all uninitialized components with the value 1.
  For example,

      dim3  Dimensions_of_block(64, 64);

  specifies a block that is 64 by 64 by 1 (because the z member is set to 1.)

  ALGORITHM
  This code is an implementation of matrix multiplication that does not take
  advantage of the shared memory within each SM (streaming multiprocessor).
  There will be a thread for each cell of the C result matrix.

  Let A be an M by N matrix and B, a N by R matrix. We assume that M,N, and R
  are divisible by 16, the block size for this implementation.

  The A and B matrices are copied from host memory to device memory. There,
  a kernel is run on a grid of blocks of threads, each of which computes a
  single cell of C.  To compute a single cell of C, C[[i,j], it has to read
  the ith row of A and the jth column of B, each of size N. Each cell does
  this. Many cells read the ith row of A and many read the jth row of B, but
  only one does both of these uniquely.

  To be clear, each row of A is read R times (for each column of B) and
  each column of B is read M times (for each row of A).

  Therefore, A is read R times from global memory and B is read M times.
```

```
  The algorithm relies on the fact that matrices are stored in row-major order:
  M(row, col) = *(M.elements + row * M.width + col)

  Functions from CUDA used here include:

    cudaError_t cudaMalloc(void** devPtr, size_t count )
    This allocates count bytes of linear memory on the GPU and returns
    in *devPtr a pointer to the allocated memory. The allocated memory
    is suitably aligned for any kind of variable. The memory is not zeroed.
    cudaMalloc()
    returns cudaErrorMemoryAllocation in case of failure.
          cudaSuccess  on success.

    cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                      enum cudaMemcpyKind kind)
    This copies count bytes from the memory area pointed to by src to the
    memory area pointed to by dst, where kind is one of
      cudaMemcpyHostToHost,
      cudaMemcpyHostToDevice,
      cudaMemcpyDeviceToHost, or
      cudaMemcpyDeviceToDevice,
    and specifies the direction of the copy. The memory areas may not overlap.
    Calling cudaMemcpy() with dst and src pointers that do not match the
    direction of the copy results in an undefined behavior.
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include "matrixmult.h"

#define BLOCK_SIZE 16

/* Matrix multiplication - Host code
  Matrix dimensions are assumed to be multiples of BLOCK_SIZE
*/
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    size_t size;

    // Declare the matrix that will reside on the device (the GPU)
    Matrix d_A;
    d_A.width = A.width;
    d_A.height = A.height;

    /* Calculate its size in bytes */
    size = A.width * A.height * sizeof(float);

    /* Next allocate space for d_A on the device. This uses the function
       cudaMalloc(). It fills d_A.elements with the location of the start
       of storage for d_A.  */
    cudaMalloc(&d_A.elements, size);

    /* Next, copy the array from the host memory to the device memory
       using cudaMemCpy(). */
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
```

```
    /* Repeat the above steps for the matrix B */
    Matrix d_B;
    d_B.width = B.width;
    d_B.height = B.height;

    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

    /* Repeat almost all of the steps for the C matrix. We do not copy it
       from host memory because it gets computed on the device and sent back
       to the host.
    */
    Matrix d_C;
    d_C.width = C.width;
    d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    /* It is time to invoke the kernel on the GPU. The kernel is run by
       several streaming multiprocessors. The C result matrix is decomposed
       into square blocks of size BLOCK_SIZE by BLOCK_SIZE. Each of these
       is computed by a block of threads. There will be a grid of such blocks.
       If the B matrix is R cells wide and the A matrix is M cells high, then
       C is M by R. Assuming R and M are multiples of BLOCK_SIZE, the grid
       is of width R/BLOCK_SIZE and of height M/BLOCK_SIZE. The next two
       instructions define the block and grid dimensions.
    */
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);

    /* Now the kernel is invoked on this grid of blocks of threads */
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    /* When the kernel completes, we read C from device memory back to
       host memory.
    */
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    /*
      cudaError_t cudaFree(void* devPtr)
      This frees the memory space pointed to by devPtr, which must have been
      returned by a previous call to cudaMalloc() or cudaMallocPitch().
      Otherwise, or if cudaFree(devPtr) has already been called before,
      an error is returned. If devPtr is 0, no operation is performed.
    */
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

void initMatrix( Matrix *M, float value)
{
    int i,j;

    M->width = 256;
    M->height= 256;

    M->elements = (float*) calloc(M->width * M->height, sizeof(float) );
    if ( M->elements == NULL )
```

```
        exit(1);
    for ( i = 0; i < M->height; i++ )
        for ( j = 0; j < M->width; j++ )
            M->elements[j + i*M->width] = value;
}


void printMatrix( Matrix M )
{
    int i,j;

    for ( i = 0; i < M.height; i++ ) {
        for ( j = 0; j < M.width; j++ )
            printf("%3.2f " , M.elements[j + i*M.width]);
        printf("\n");
    }
}

int main(int argc, char* argv[])
{
    Matrix A, B, C;

    initMatrix(&A, 2.0);
    initMatrix(&B, 3.0);
    initMatrix(&C, 0.0);

    MatMul ( A, B,  C );
    printMatrix(C);
    return 0;
}
```