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

  CUDA-based MATRIX MULTIPLICATION

  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.

  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 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 takes advantage
  of the shared memory within each SM (streaming multiprocessor). It is more
  complex than the algorithm that uses just global device memory, for several
  inter-related reasons.

  Each thread block is responsible for computing one square sub-matrix Csub of
  the product matrix C, and each thread within the block is responsible for
  computing one element of Csub.

  Csub is equal to the product of two rectangular matrices:
    the sub-matrix of A of dimension A_width x BLOCK_SIZE that has the same
    row indices as Csub, and
    the sub-matrix of B of dimension BLOCK_SIZE x A_width that has the same
    column indices as Csub.

  In order to fit into the device's resources, these two rectangular matrices
  are divided into as many square matrices of dimension BLOCK_SIZE as necessary
  and Csub is computed as the sum of the products of these square matrices.

  Each of these products is computed by first loading the two corresponding
```

```
   square matrices from global memory to shared memory, having each
   thread compute one element of the product, and writing the resulting square
   sub-matrix back to global memory afterward.

   To make the copying from global to shared memory efficient, each thread is
   responsible for copying a single element from each of the A and B matrices.
   The copying is done in such a way to maximize the memory bandwidth, which will
   be explained below.
*/

#define BLOCK_SIZE 16   // Thread block size


/* The __global__ qualifier declares a function as being a kernel.
   A kernel is a function that is executed on the device (the GPU), and
   is callable from the host (CPU).
   This is a forward declaration of the device multiplication function
*/
__global__ void Muld(float*, float*, int, int, float*);


/** Multiply(&A, %B, hA, wA, wB, &C)  Host multiplication function, performs
 *  C = A * B, where
 *  hA is the height of A
 *  wA is the width of A
 *  wB is the width of B
 *  and A, B, and C are linear arrays allocated by the calling program.
 *  The arrays must have been allocated correctly; no chek is made.
 */
void Multiply ( const float  *A,
                const float  *B,
                int          height_A,
                int          width_A,
                int          width_B,
                float         *C)
{
    int size;

    /* Copy matrices A and B to the device memory
       To copy requires using cudaMemCpy, which can copy either from host
       to device, from device to host, or from device to device.
    */
    float* A_on_device;
    size = height_A * width_A * sizeof(float);
    cudaMalloc((void**)&A_on_device, size);
    cudaMemcpy(A_on_device, A, size, cudaMemcpyHostToDevice);


    float* B_on_device;
    size = width_A * width_B * sizeof(float);
    cudaMalloc((void**)&B_on_device, size);
    cudaMemcpy(B_on_device, B, size, cudaMemcpyHostToDevice);

    // Allocate matrix C on the device
    float* C_on_device;
    size = height_A * width_B * sizeof(float);
    cudaMalloc((void**)&C_on_device, size);

    /* Compute the execution configuration assuming
       the matrix dimensions are multiples of BLOCK_SIZE
```

```
     The dim3 declaration is used here. This specifies that dimBlock
     is BLOCK_SIZE x BLOCK_SIZE x 1
   */
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

   /* width_B/dimBlock.x is the length of a row of B divided by the horizontal
      size of a block, which yields the number of blocks in the horizontal
      dimension of the Grid.
      Similarly,  height_A/dimBlock.y is the height of A divided by the vertical
      size of a block, which is the number of blocks vertically in the grid.
      These two values define the shape of the grid, i.e. the number of
      blocks horizontally and vertically. */
   uint gridwidth  = width_B / dimBlock.x;
   uint gridheight = height_A / dimBlock.y;
   dim3 dimGrid(gridwidth, gridheight);

   /* Launch the kernel now.
      The syntax
         func<<< Dg, Db, Ns, S >>>(parameter_list)
      defines a kernel function with a parameter list that is executed on a
      specific configuration defined as follows:
         Dg is of type dim3 and specifies the dimension and size of
         the grid; Dg.x * Dg.y equals the number of blocks being launched;
         Dg.z must be equal to 1;
         Db is of type dim3 and specifies the dimension and size of
         each block, such that Db.x * Db.y * Db.z equals the number of
         threads per block;
         Ns is of type size_t and specifies the number of bytes in shared
         memory that is dynamically allocated per block for this call
         in addition to the statically allocated memory; Ns is an
         optional argument which defaults to 0;
         S is of type cudaStream_t and specifies the associated stream;
         S is an optional argument which defaults to 0.

      The Muld() function will be run by every thread in every block of
      the grid. Thread blocks execute independently: they execute in any
      order, in parallel or in series, unpredictably. The function must be
      correct regardless of the order.
   */
   Muld<<<dimGrid, dimBlock>>>(A_on_device,
                               B_on_device,
                               width_A,
                               width_B,
                               C_on_device);

   // Copy result C from the device to host memory
   cudaMemcpy(C, C_on_device, size, cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(A_on_device);
   cudaFree(B_on_device);
   cudaFree(C_on_device);
}

  /*  Note again the __global__ qualifier: this is a kernel function
      This also means that every thread executes this function.
      When a thread executes this function, it has a specific thread id
      and block id. The thread id is the value of threadIdx, used below,
      and the block id is stored in blockIdx, used below.
      threadIdx  and blockIdx are of type unit3.
   */
__global__ void Muld(float* A, float* B, int width_A, int width_B, float* C)
{
   // Block index
   int block_col  = blockIdx.x;
   int block_row  = blockIdx.y;

   // Thread index
   int thread_col = threadIdx.x;
   int thread_row = threadIdx.y;

   /* Index of the first cell of the sub-matrix of A processed by the block
      of threads. There are width_A cells in a row and BLOCK_SIZE many rows
      in each sub-matrix. Therefore,
         width_A * BLOCK_SIZE * block_row
      is the total number of cells from the start of A to the leftmost cell
      of the first row of the horizontal set of sub-matrices of A processed
      by this block of threads.
   */
   int aBegin = width_A * BLOCK_SIZE * block_row;

   /* The index of the last cell in the row of A that starts at ABegin is
      aBegin plus width_A-1, since a row is width_A cells long.
   */
   int aEnd = aBegin + width_A - 1;

   /* Step size used to iterate through the sub-matrices of A
      The upper left corner of the next block of A is BLOCK_SIZE columns
      from the current block's corner, so the increment is just BLOCK_SIZE.
   */
   int aStep = BLOCK_SIZE;

   /* Index of the first sub-matrix of B processed by the block */
   int bBegin = BLOCK_SIZE * block_col;

   /* Step size used to iterate through the sub-matrices of B
      The upper left corner of the next block of B is BLOCK_SIZE rows
      below the upper left corner of the current block. Each row has
      width_B bytes, so the increment is BLOCK_SIZE * width_B. */
   int bStep = BLOCK_SIZE * width_B;

   /* The element of the block sub-matrix that is computed by the thread */
   float Csub = 0;

   /* Loop over all the sub-matrices of A and B required to
      compute the block sub-matrix */
   for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {

      /* The __shared__ qualifier declares a variable that
          * resides in the shared memory space of a thread block,
          * has the lifetime of the block, and
          * is only accessible from the threads within the block.
         The As and Bs matrices declared below are in the shared memory of
         the block; As is for the sub-matrix of A, and Bs, a sub-matrix of B.
      */
      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

      /* Shared memory for the sub-matrix of B */
```

```
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

/* The next step loads the matrices from global memory to shared memory;
   each thread loads one element of each matrix. There are two things
   to discuss about this: the logic and the performance.

   Performance Issue:
   The global memory access by all threads of a half-warp is coalesced
   into one or two memory transactions if it satisfies the following
   three conditions:
     1. Threads must access either
        4-byte words, resulting in one 64-byte memory transaction,  or
        8-byte words, resulting in one 128-byte memory transaction, or
        16-byte words, resulting in two 128-byte memory transactions;
     2. All 16 words must lie in the same segment of size equal to the
        memory transaction size (or twice the memory transaction size
        when accessing 16-byte words);
     3. Threads must access the words in sequence, meaning that the
        kth thread in the half-warp must access the kth word.

   In the instructions below, each thread accesses a 4-byte word
   ( because it is a float) and 16 threads do this simultaneously,
   for a total of 64-bytes in each transaction. The words accessed by
   the threads are in sequence, so the memory accesses satisfy the
   conditions stated above, which implies that the accesses are
   coalesced and will require just two trips to global memory.

   Logic:
   The matrices As and Bs are shared by all threads in a given block.
   There are multiple blocks in a grid and hence multiple shared
   matrices at any given time. We focus on just one pair of these.
   Initially, the loop index variable, a, points to the upper-left
   corner of the rectangular sub-matrix of A that will contribute to
   the CSub result, and b points to the upper-left corner of the
   B sub-matrix.

   Remember that A and B are linear arrays in row-major order. The
   cell of A needed by the thread with id (row,col) is at an offset
   from a of width_A*row + col.  The cell of B is similarly
   at an offset of width_B*row + col. The thread copies these two
   values into As[row,col] and Bs[rox,col] respectively.

   All threads in the thread block do this simultaneously, so the two
   arrays, As and Bs, are filled by the collection of threads in the
   thread block. These two arrays have the contents of just one
   BLOCK_SIZE by BLOCK_SIZE sub-matrix of each of A and B, not an
   entire row or column od A and B.

   In the next iteration of the loop, the a and b index variables
   point to the upper-left corners of two difference sub-matrices
   of A and B. The a index points to the sub-matrix to the immediate
   right of the preceding one, and b, to the one imediately below
   the preceding one.

   This continues until every sub-matrix in the row of A and in the
   column of B have been copied into As and Bs, one after the other.
*/
As[thread_row][thread_col] = A[a + width_A * thread_row + thread_col];
Bs[thread_row][thread_col] = B[b + width_B * thread_row + thread_col];
```

```
/* The next step synchronizes to make sure the matrices are loaded:
   __syncthreads() is a barrier synchronization call; all threads in
   a single block wait here until every thread has made the call,
   at which point it returns in each thread. This guarantees that the
   BLOCK_SIZE by BLOCK_SIZE matrices As and Bs are completely filled.
*/
__syncthreads();

/* The next step is to compute the inner product of the row of As
   and column of Bs assigned to this thread, As[thread_row],
   and Bs[thread_col], i.e., the sum

     As[i][0]*Bs[0][j] + As[i][1]*Bs[1][j] + ... As[i][N-1]Bs[N-1][j]

   The loop below,
      for (int k = 0; k < BLOCK_SIZE; ++k)
         Csub += As[thread_row][k] * Bs[k][thread_col];
   adds to Csub the inner product of two vectors of size 16
   corresponding to the row of a sub-matrix of A and a column of a
   sub-matrix of B. But each time the outer for-loop steps the index
   a to the next sub-matrix, this inner product is continued for the
   next sub-matrix to the right in A and the one below in B, so that
   when the loop completes, the inner product of the entire row of A
   and column of B has been computed and stored in Csub.

   For a device with compute capability 1.x, there are 16 banks in
   shared memory, each 4-bytes wide. A float is stored in one word
   of a bank.

   Remember that all threads within a single warp execute the same
   instruction at a time. Therefore, when a kernel has a loop such as
      for (int k = 0; k < BLOCK_SIZE; ++k)
         Csub += As[thread_row][k] * Bs[k][thread_col];

   each thread executes the instruction in the body simultaneously.
   Consider the assignment statement,

      Csub += As[thread_row][k] * Bs[k][thread_col];

   for fixed k. Each of the threads in a half-warp have the same
   value of thread_row, because a warp consists of 16 threads in
   row i followed by 16 threads in row i+1, for each i = 0,2,4,... 30.
   So the first half-warp accesses only row i, and the second half-warp
   accesses only row i+1. Put another way, all threads in a half-warp
   have the same value of thread_row. Thus, for a fixed value of k,
   As[thread_row][k] is the same memory location in each thread in a
   half-warp and will be read once and stored locally for each thread
   in a register.

   The values Bs[k][thread_col] are 16 successive 4-byte words,
   (because the arrays are stored in row-major order), stored
   in banks 0, 1, 2, ..., 15 of the shared memory. This is a fact of
   how shared memory is used by the GPU: successive 4-byte words are in
   successive banks. Therefore there is no bank conflict in the threads
   accessing these banks simultaneously. When simultaneous memory
   requests are to different banks, they are accessed concurrently.
   Therefore, the values Bs[k][thread_col] accessed by the 16 threads
   in a half-warp are accessed in just two clock cycles.
*/
for (int k = 0; k < BLOCK_SIZE; ++k)
```

```
            Csub += As[thread_row][k] * Bs[k][thread_col];

        /* Synchronize to make sure that the preceding computation is done
           before loading two new sub-matrices of A and B in the next iteration
        */
        __syncthreads();
    }

    /* Now that the loop has terminated, the value in Csub has to be copied
       back to global memory, to the corresponding cell of the C matrix.
       Each thread writes one element. The position is relative to the
       upper-left corner element of the C sub-matrix computed by this
       thread block. This sub-matrix is in position [block_row, block_col],
       so the upper-left corner is in the row given by
            width_B*BLOCK_SIZE*block_row
       at an offset from the 0 collumn of
            BLOCK_SIZE*block_col.
    */
    int c = width_B * BLOCK_SIZE * block_row + BLOCK_SIZE * block_col;

    /* The cell within this sub-matrix is in the row of the sub-matrix
       given by  width_B*thread_row, and column thread_col. We assign
       Csub to this cell in the global array.
    */
    C[c + width_B * thread_row + thread_col] = Csub;
}
```