```
/ Thread block size
#define BLOCK_SIZE 16

// Forward declaration of device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B,
         int hA, int wA, int wB, float* C) {

  int size;
  // Load A and B to the device
  float* Ad;
  size = hA * wA * sizeof(float);
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  float* Bd;
  size = wA * wB * sizeof(float);
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
  // Allocate C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);
  // Compute execution configuration assuming the
  // matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B,
                     int wA, int wB, float* C) {

  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;
  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  // Index of first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;
  // Index of the last sub-matrix of A processed by the block
  int aEnd = aBegin + wA - 1;
  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;
  // Index of first sub-matrix of B processed by the block
  int bBegin = BLOCK_SIZE * bx;
  // Step size used to iterate through the sub-matrices of B
  int bStep = BLOCK_SIZE * wB;
  // Element of the block sub-matrix computed by the thread
  float Csub = 0;
  // Loop over A & B sub-matrices computing block sub-matrices
  for (int a = aBegin, b = bBegin;
       a <= aEnd; a += aStep, b += bStep) {
    // Shared memory for the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load the matrices from global memory to shared memory
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    // Synchronize to make sure the matrices are loaded
    __syncthreads();
    // Multiply the two matrices together; each
    // thread computes one element of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    // Synchronize before loading two new sub-matrices
    __syncthreads();
  }
  // Write the block sub-matrix to global memory
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```