Le code ci-dessous calcul la multiplication entre deux matrices carrées – dont la taille est passée en paramètre. Le calcul s'effectue sur le GPU à l'aide du framework JCuda. Le kernel (méthode s'effectuant sur le CPU) est dans un fichier à part.
Après avoir pris connaissance du code, répondez aux 3 questions ci-dessous en cochant les cases.
***Note** : il n'est pas nécessaire de comprendre l'algorithme utilisé, mais plutôt l'utilisation du GPU.*

1. Quel est votre niveau d'expérience en programmation ?

Très grande expérience en programmation	1	2	3	4	5	Aucune expérience en programmation

2. Quel est notre niveau d'expérience en programmation GPU ?

Très grande expérience en programmation GPU	1	2	3	4	5	Aucune expérience en programmation GPU

3. Comment jugez-vous la difficulté à comprendre l'utilisation du GPU dans le code ?

Très difficile à comprendre	1	2	3	4	5	Très facile à comprendre

```
 1 import java.util.Random;
 2 import jcuda.*;
 3 import jcuda.runtime.*;
 4 import jcuda.driver.*;
 5 import static jcuda.driver.JCudaDriver.*;
 6
 7 public class GPMatrix {
 8
 9   public static int verbose = 0; // level 0,1,2
10   public static int size;
11
12   public static void main(String[] args) {
13
14     // Params handling
15     if(args.length < 1) {
16       System.err.println("Needs size argument");
17       System.exit(0);
18     }
19     size = Integer.valueOf(args[0]);
20
21     // Matrix declaration
22     double[] A = new double[size * size];
23     double[] B = new double[size * size];
24     double[] C = new double[size * size];
25
26     matrixInit(A,B);
27
28
29     // Enable exception
30     JCudaDriver.setExceptionsEnabled(true);
31
32     // Initialize the driver and create a context for the first device
33     cuInit(0);
34     CUdevice device = new CUdevice();
35     cuDeviceGet(device, 0);
36     CUcontext context = new CUcontext();
37     cuCtxCreate(context, 0, device);
38
39     // Load the CUDA kernel ptx file
40     CUmodule module = new CUmodule();
41     cuModuleLoad(module, "../JCudaMatrix.ptx");
42
43     // Obtain a function pointer to the kernel function
44     CUfunction function = new CUfunction();
45     cuModuleGetFunction(function, module, "mul");
46
```

```
47      // Allocate the device input data, and copy the host input data
48      // to the device
49      int ptrSize = size * size * Sizeof.DOUBLE;
50      CUdeviceptr deviceInputA = new CUdeviceptr();
51      cuMemAlloc(deviceInputA, ptrSize);
52      cuMemcpyHtoD(deviceInputA, Pointer.to(A), ptrSize);
53
54      CUdeviceptr deviceInputB = new CUdeviceptr();
55      cuMemAlloc(deviceInputB, ptrSize);
56      cuMemcpyHtoD(deviceInputB, Pointer.to(B), ptrSize);
57
58      CUdeviceptr deviceOutput = new CUdeviceptr();
59      cuMemAlloc(deviceOutput, ptrSize);
60
61      // Set up the kernel parameters: A pointer to an array
62      // of pointers which points to the actual values
63      Pointer kernelParameters = Pointer.to(
64        Pointer.to(deviceInputA),
65        Pointer.to(deviceInputB),
66        Pointer.to(deviceOutput),
67        Pointer.to(new int[]{size})
68      );
69
70      // Kernel call parameters
71      int blockSizeX = 256;
72      int gridSizeX = (int)Math.ceil((double)size / blockSizeX);
73
74      // -------------------------------
75      long startTime = System.currentTimeMillis();
76      // --- Start of benchmark zone --->
77      cuLaunchKernel(function,
78        gridSizeX, 1, 1,       // Grid dimension
79        blockSizeX, 1, 1,      // Block dimension
80        0, null,               // Shared memory size and stream
81        kernelParameters, null // Kernel and extra parameters
82      );
83      cuCtxSynchronize();
84      // <--- End of benchmark zone -----
85      long stopTime = System.currentTimeMillis();
86      // -------------------------------
87
88      // Benchmark time elapsed computation
89      long elapsedTime = stopTime - startTime;
90
91      // Output
92      if(verbose > 0) System.out.println("Time elapsed: " + elapsedTime/1000 + "s");
93      System.out.println(elapsedTime/1000);
94
95    }
96
97    // Matrix random initialization of A and B
98    public static void matrixInit(double[] A, double[] B) {
99      Random r = new Random();
100     for(int row=0; row < size; row++) {
101       for(int col=0; col < size; col++) {
102         A[ (row*size)+col ] = r.nextDouble();
103         B[ (row*size)+col ] = r.nextDouble();
104       }
105     }
106   }
107
108
109 }
110
```

Fichier contenant le kernel (méthode s'exécutant sur le GPU) :

```
extern "C"
__global__ void mul(double* A, double* B, double* C, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < size) {
    // compute a column
    for(int j=0; j < size; j++) {
      double sum = 0.0;
      for(int k=0; k < size; k++) {
        sum += A[ (i*size)+k ] * B[ (k*size)+j ];
      }
      C[ (i*size)+j ] = sum;
    }
    // end of column computing
  }
}
```