TP 7 Java natif avec JNI, FFM et GraalVM

Accès natif progressif : du simple au complexe, dans un contexte fonctionnel

Objectif : montrer comment Java dialogue avec du C dans des scénarios réalistes (calcul vectoriel, convolution 1D, statistiques...).

Java natif gagne un rôle stratégique dans plusieurs domaines technologiques où chaque milliseconde, chaque mégaoctet et chaque watt comptent. Dans le cloud, transformer une application Java en binaire natif réduit drastiquement le temps de démarrage : un service qui prendrait plusieurs centaines de millisecondes à s'initialiser sur la JVM démarre parfois en moins de dix millisecondes en natif. Cet écart change toute la dynamique du serverless, du scaling automatique et des microservices éphémères. En intelligence artificielle, les bibliothèques d'inférence doivent souvent appeler du code bas niveau (BLAS, CUDA, ONNX Runtime...). La compilation native permet une intégration plus directe, limite les surcoûts d'abstraction et facilite l'optimisation liée au matériel (vectorisation, AVX, GPU, NPU). Dans l'embarqué et l'IoT, l'équation est encore plus serrée : processeurs modestes, mémoire limitée, énergie rare. Une application Java compilée en natif se comporte comme un programme C autonome, fonctionne sans JVM, démarre instantanément et consomme peu. Cela ouvre la porte à une écriture haut niveau, sûre et portable des couches applicatives, tout en gardant la finesse d'accès au matériel grâce aux appels natifs et à l'API FFM. L'ensemble forme une trajectoire cohérente : Java reste expressif et productif, mais devient capable d'opérer dans les environnements où la légèreté, la rapidité et l'efficacité matérielle ne sont plus négociables.

---

1. Bibliothèque C : module "signal & math"

JNI, la passerelle historique, reste le "tuyau brut" entre Java et C. C'est un peu le métier d'archéologue logiciel : beaucoup de cérémonies, beaucoup de fichiers, beaucoup de soin à apporter à la mémoire. Mais c'est extrêmement puissant. JNI donne un accès direct au monde natif sans ambiguïté, et pour certaines bibliothèques existantes (BLAS, FFTW, OpenCV, CUDA), c'est encore la voie royale. On y sacrifie un peu de confort, mais on gagne en contrôle total. Dans un sens, JNI est le contrat de paix originel entre l'univers Java et le reste du monde.

1.1 Fonctions simples

```
double hyp(double a, double b) {
    return sqrt(a*a + b*b);
}
```

1.2 Moyenne et variance d'un tableau

```
double avg(double *data, int n) {
    double s = 0;
    for(int i=0;i<n;i++) s += data[i];
    return s / n;
}

double var(double *data, int n) {
    double m = avg(data, n);
    double s = 0;
    for(int i=0;i<n;i++) {
        double d = data[i] - m;
        s += d * d;
    }
    return s / n;
}
```

1.3 Convolution 1D

```
void convolve(const double *signal, int n,
              const double *kernel, int k,
              double *out) {
    for (int i = 0; i < n; i++) {
        double s = 0;
        for (int j = 0; j < k; j++) {
            int idx = i - j;
            if (idx >= 0)
                s += signal[idx] * kernel[j];
        }
```

```
        out[i] = s;
    }
}
```

Compile:

```
gcc -shared -fPIC -o libsig.so sig.c -lm
```

---

2. Java 8 — JNI

2.1 Signature Java

```
public class SigJNI {
    static {
        System.loadLibrary("sig");
    }

    public static native double hyp(double a, double b);

    public static native double avg(double[] data);
    public static native double var(double[] data);

    public static native double[] convolve(double[] signal, double[] kernel);
}
```

2.2 Header JNI

```
javac SigJNI.java
javah SigJNI
```

2.3 Implémentation C

Exemple pour avg :

```
JNIEXPORT jdouble JNICALL
Java_SigJNI_avg(JNIEnv *env, jclass cls, jdoubleArray arr) {
    int n = (*env)->GetArrayLength(env, arr);
    jdouble *ptr = (*env)->GetDoubleArrayElements(env, arr, 0);
    double result = avg(ptr, n);
    (*env)->ReleaseDoubleArrayElements(env, arr, ptr, 0);
    return result;
}
```

Convolution :

```
JNIEXPORT jdoubleArray JNICALL
Java_SigJNI_convolve(JNIEnv *env, jclass cls,
                     jdoubleArray s, jdoubleArray k) {
    int ns = (*env)->GetArrayLength(env, s);
    int nk = (*env)->GetArrayLength(env, k);

    jdouble *ps = (*env)->GetDoubleArrayElements(env, s, 0);
    jdouble *pk = (*env)->GetDoubleArrayElements(env, k, 0);

    jdoubleArray out = (*env)->NewDoubleArray(env, ns);
    jdouble *po = malloc(sizeof(double)*ns);

    convolve(ps, ns, pk, nk, po);

    (*env)->SetDoubleArrayRegion(env, out, 0, ns, po);

    free(po);
    (*env)->ReleaseDoubleArrayElements(env, s, ps, 0);
```

```
    (*env)->ReleaseDoubleArrayElements(env, k, pk, 0);
    return out;
}
```

---

3. Java 25 — FFM (Foreign Function & Memory)

L'API FFM (Foreign Function & Memory), elle, s'avance comme l'évolution naturelle. Elle retire les archaïsmes de JNI, apporte un modèle mémoire sûr, élimine les contorsions liées à JNIEnv et permet de manipuler des segments comme des blocs à la fois modernes et explicites. C'est la démocratisation de l'accès natif : appeler une fonction C devient presque aussi trivial que manipuler une lambda. L'API FFM est aussi pensée pour l'optimisation : alignement mémoire, accès direct, efficacité sur architectures modernes... tout cela sans perdre la robustesse du monde Java. Sur des projets IA, cloud ou IoT, FFM devient souvent le bon équilibre entre expressivité et efficacité.

Objectif : appeler les mêmes fonctions C **sans JNI**, avec segments mémoire sécurisés.

Chargement :

```
System.loadLibrary("sig");

Linker linker = Linker.nativeLinker();
SymbolLookup lookup = SymbolLookup.loaderLookup();
```

3.1 Appel de avg (tableaux Java → segments C)

```
double[] data = {1,2,3,4,5};

try (Arena arena = Arena.ofConfined()) {
    MemorySegment seg = arena.allocateArray(ValueLayout.JAVA_DOUBLE, data);

    MethodHandle mh = linker.downcallHandle(
        lookup.find("avg").orElseThrow(),
        FunctionDescriptor.of(ValueLayout.JAVA_DOUBLE,
                              ValueLayout.ADDRESS, ValueLayout.JAVA_INT)
    );

    double avg = (double) mh.invoke(seg, data.length);
    System.out.println("avg = " + avg);
}
```

3.2 Convolution 1D avec FFM

```
double[] signal = {1,2,3,4,5};
double[] kernel = {1,1};

try (Arena arena = Arena.ofConfined()) {
    var s = arena.allocateArray(ValueLayout.JAVA_DOUBLE, signal);
    var k = arena.allocateArray(ValueLayout.JAVA_DOUBLE, kernel);
    var out = arena.allocateArray(ValueLayout.JAVA_DOUBLE, signal.length);

    MethodHandle mh = linker.downcallHandle(
        lookup.find("convolve").orElseThrow(),
        FunctionDescriptor.ofVoid(
            ValueLayout.ADDRESS, ValueLayout.JAVA_INT,
            ValueLayout.ADDRESS, ValueLayout.JAVA_INT,
            ValueLayout.ADDRESS
        )
    );

    mh.invoke(s, signal.length, k, kernel.length, out);

    double[] res = out.toArray(ValueLayout.JAVA_DOUBLE);
    System.out.println(Arrays.toString(res));
}
```

4. GraalVM — Compilation native

GraalVM Native Image pousse l'expérience encore plus loin : Java sans JVM. L'application devient un binaire autonome, statiquement analysé, taillé pour le démarrage instantané et une consommation mémoire minimaliste. C'est le super-pouvoir de Java dans les environnements où chaque milliseconde est une ressource : microservices qui scalent à chaud, API serverless qui doivent répondre immédiatement, dispositifs embarqués qui n'autorisent ni JIT ni overhead inutile. La compilation native transforme Java en compagnon crédible du C et du Rust dans les environnements contraints, tout en conservant la sécurité du typage et la productivité de l'écosystème Java.

Compilation :

```
native-image -H:Name=sig-ffm SigFFM.java
```

---

5. Mini-projet final pour les élèves

Créer un module C complet :

- statistiques : moyenne, variance, médiane\
- filtrage : convolution 1D, filtre moyenneur\
- géométrie : distance, angle, produit scalaire

Puis :

1. Écrire les appels JNI\
2. Écrire les appels FFM\
3. Compiler en exécutable natif\
4. Mesurer les performances\
5. Analyser les bénéfices / limites de chaque approche
