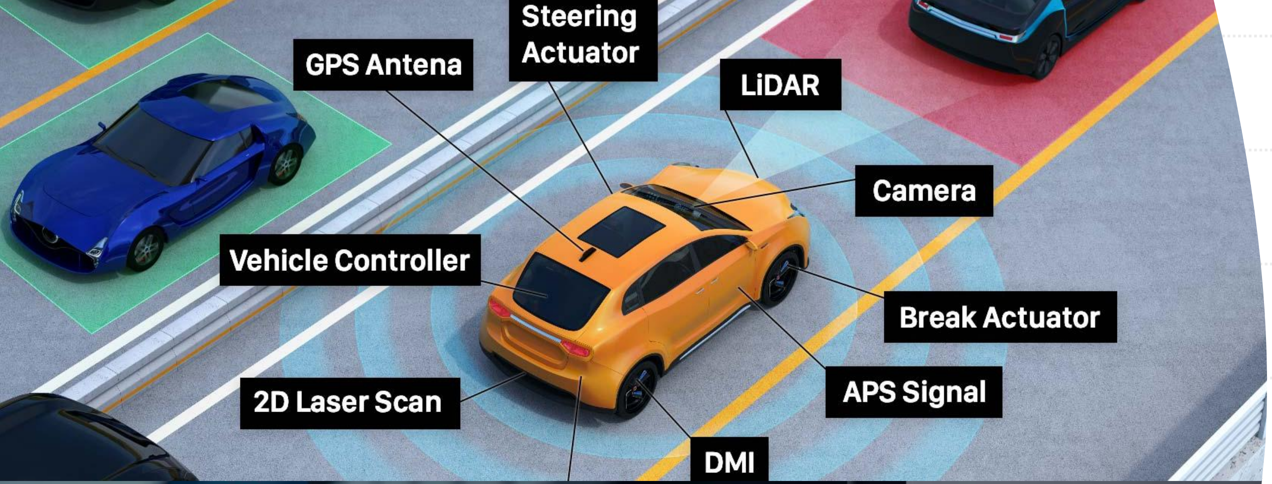
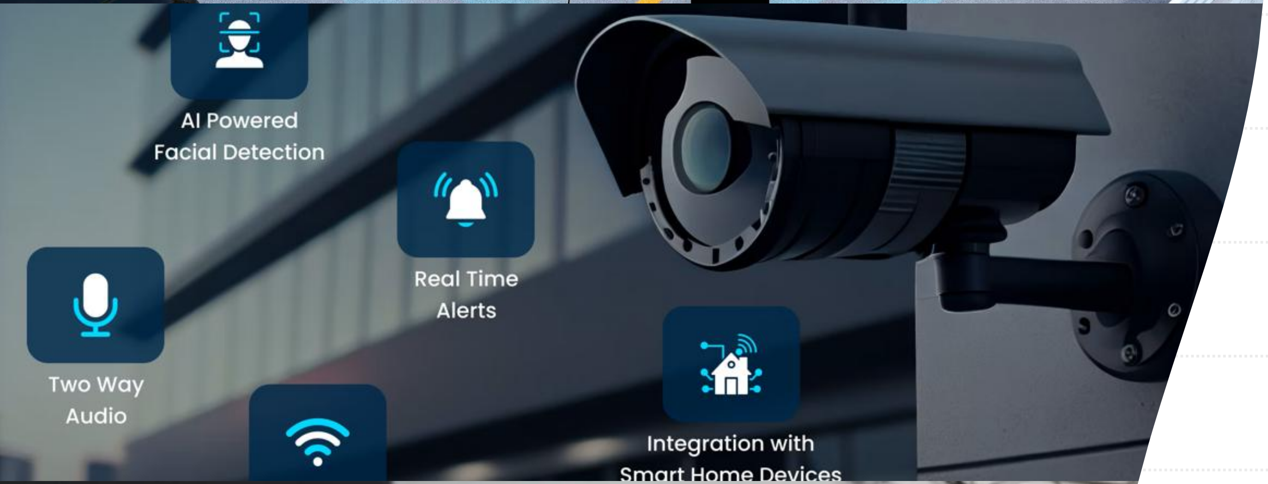
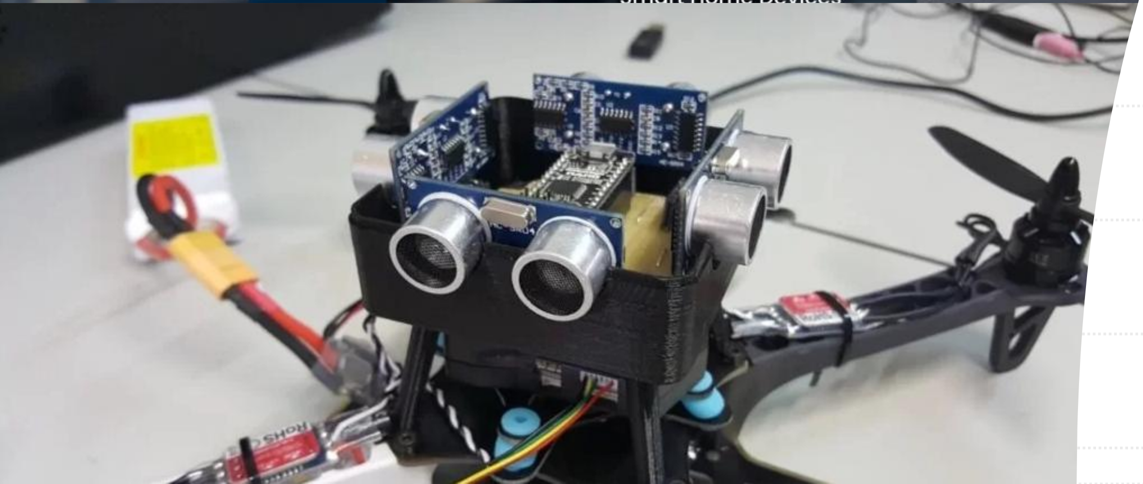
Simplificarea execuției kernel-urilor OpenCL și studiu de performanță pe diverse platforme hardware

Student: Vlad STOEAN

Profesor Îndrumător: conf. dr. ing. Andrei STAN



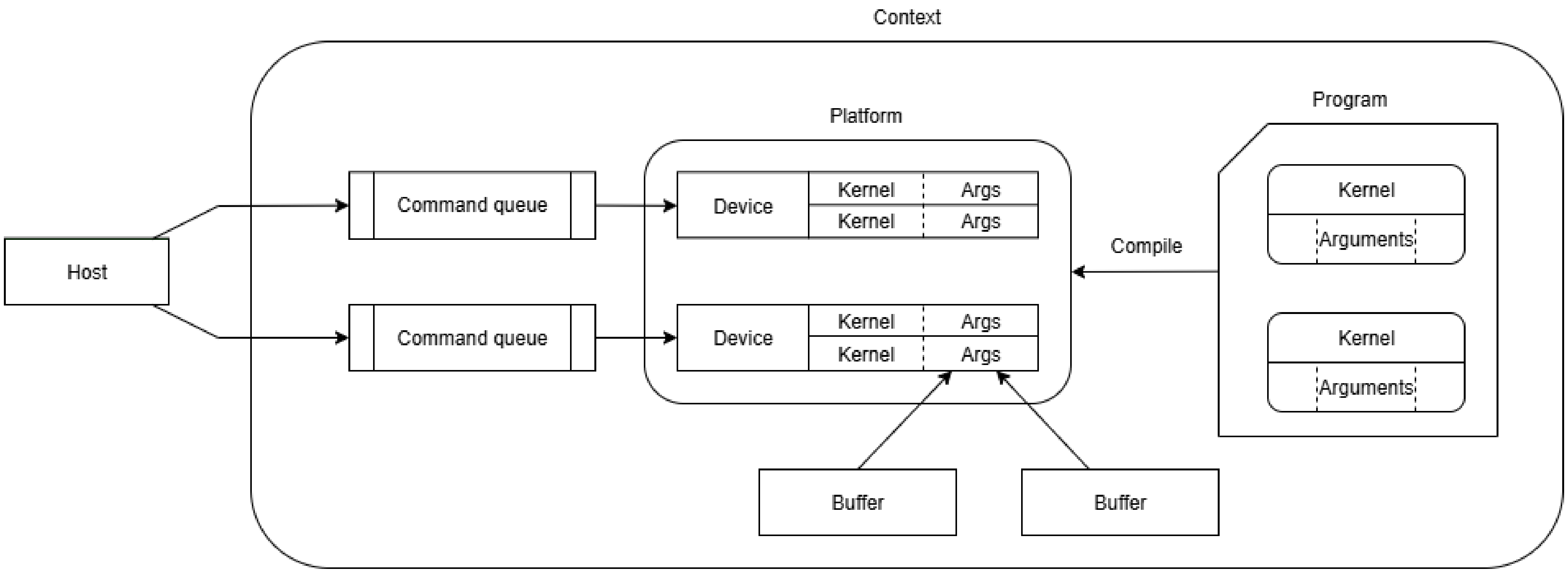




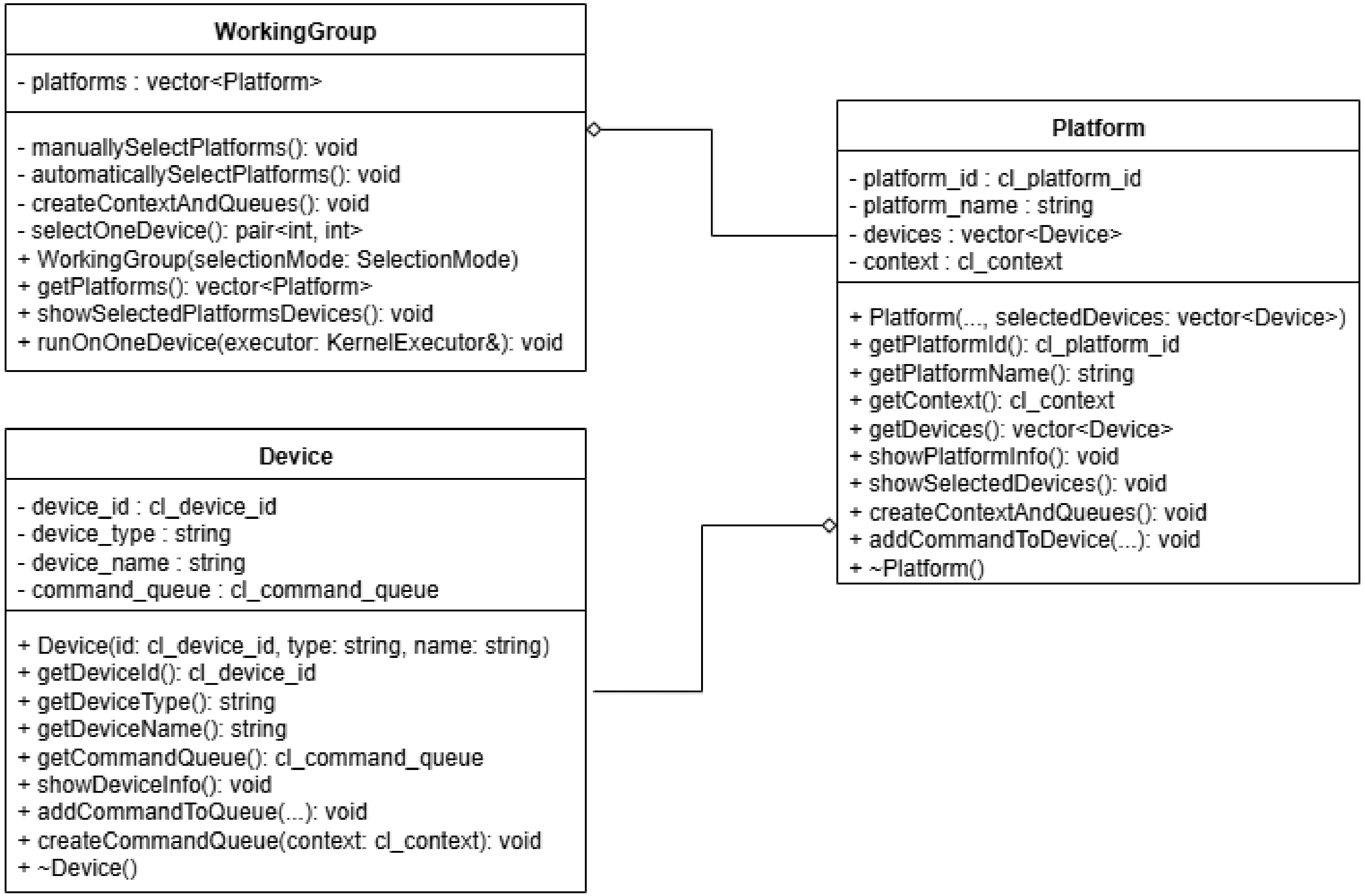
Contextul și Motivarea Temei

Acceleratoarele embedded sunt utilizate tot mai frecvent în aplicații moderne precum: mașini autonome, supraveghere video inteligentă și drone autonome. Acestea necesită procesare paralelă rapidă pentru analiză de date în timp real.

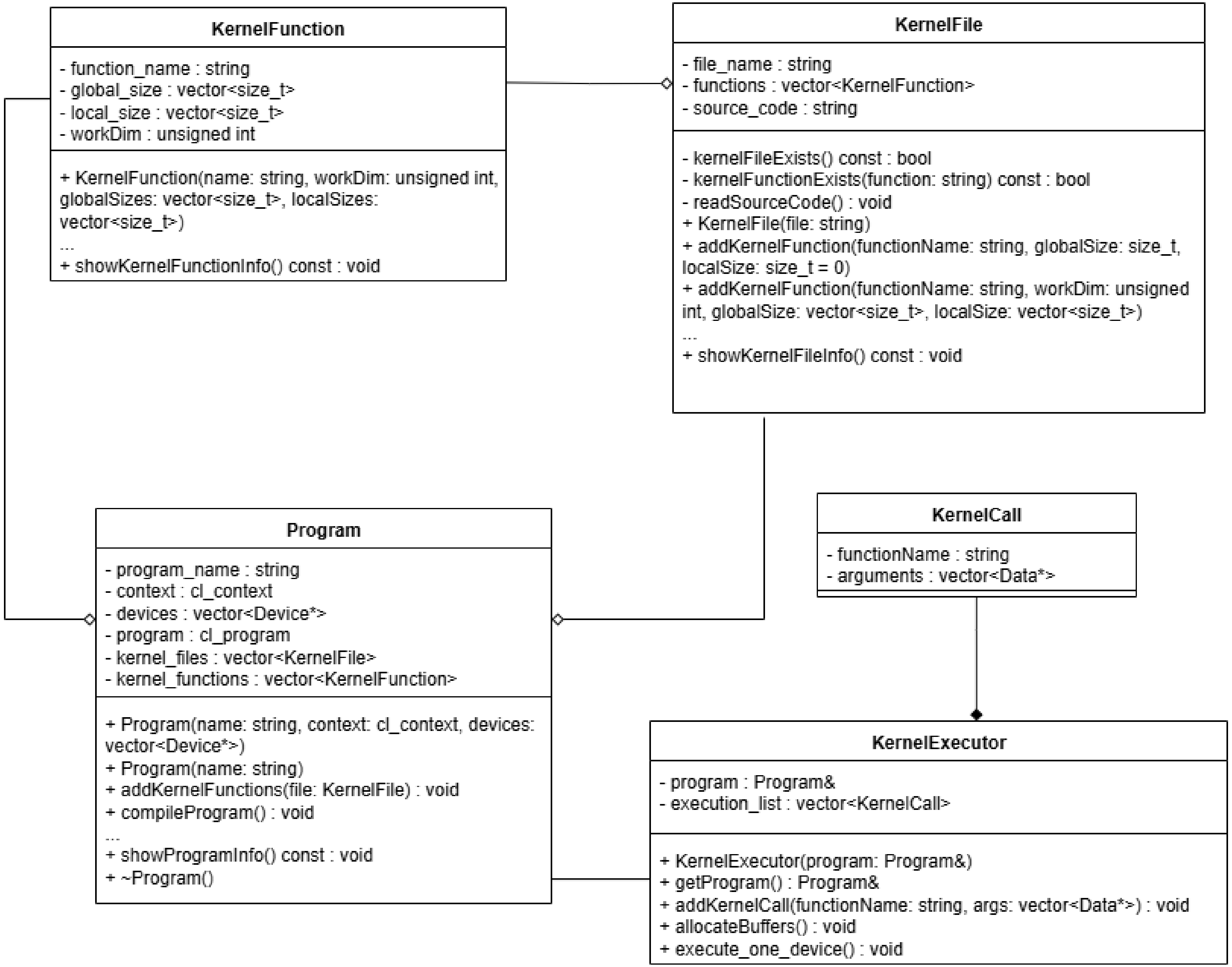
Modelul de lucru OpenCL



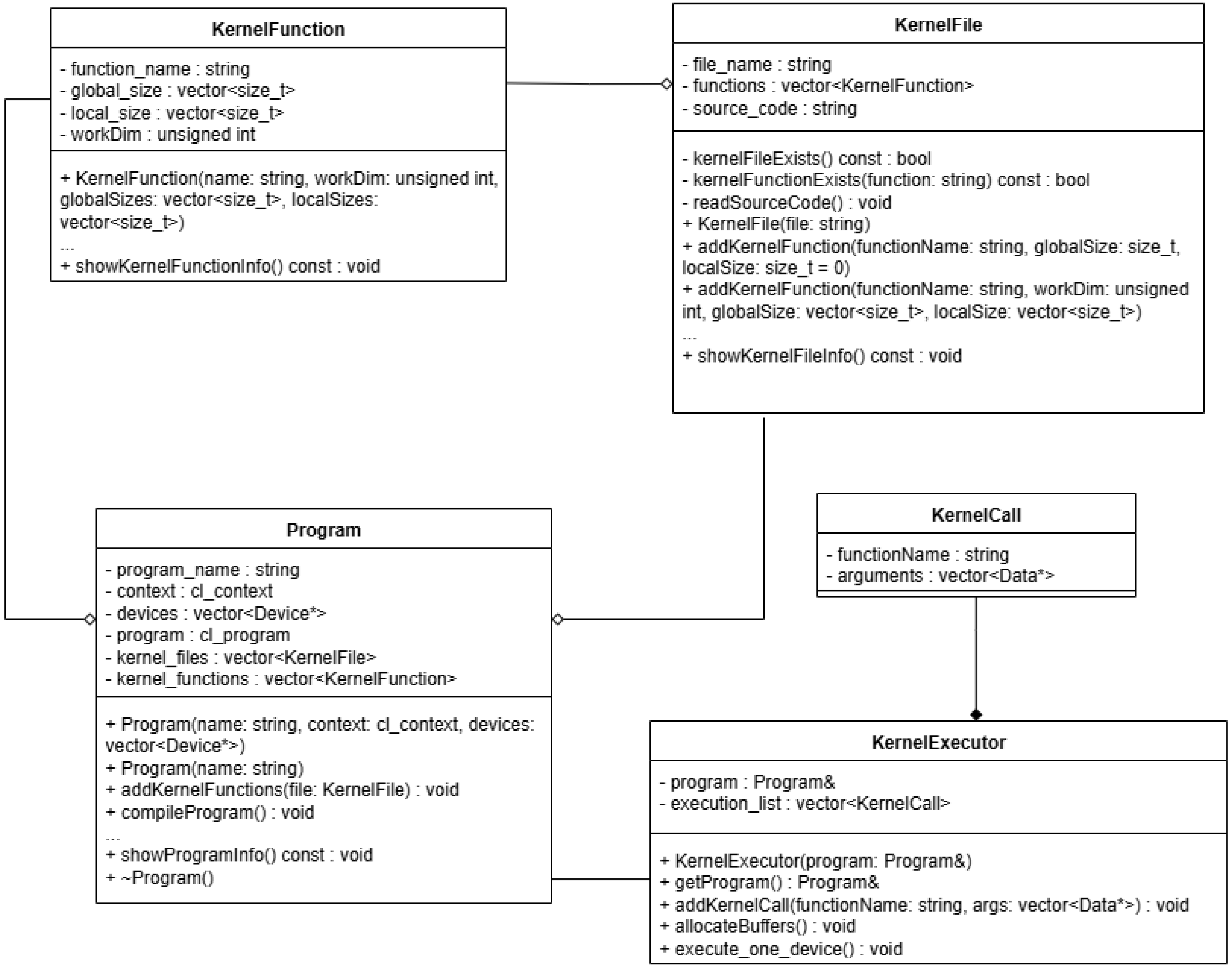
Clase pentru identificarea și alegerea resurselor hardware



Clase pentru definirea și execuția kernelurilor OpenCL



Clase pentru gestionarea argumentelor kernel-urilor



Clasă pentru scrierea și citirea datelor de test

```
Utils
--------------------------------------------------
+ readVectorFromFile<T>(parentFolder: string, filename: string) : vector<T>
+ writeVectorToFile<T>(parentFolder: string, filename: string, data: vector<T>) : void
+ readMatrixFromFile<T>(parentFolder: string, filename: string, rows: unsigned int&,
cols: unsigned int&) : vector<T>
+ generateRandomMatrixToFile(folder: string, filename: string, rows: int, cols: int) : void
```

LOC folosind framework-ul: 23

```cpp
// Device selection
WorkingGroup workingGroup(SelectionMode::Manual);

// KernelFiles
KernelFile vectorAdd("vector_add.cl");

// Host memory initialization
vector<float> a_vec = Utils::readVectorFromFile<float>("vector_add_f32", "a.txt");
vector<float> b_vec = Utils::readVectorFromFile<float>("vector_add_f32", "b.txt");

// KernelFunctions
vectorAdd.addKernelFunction("vector_add_f32", 4096);

// Program initialization
Program program("VectorOps");
program.addKernelFunctions(vectorAdd);

// OpenCL Data - Buffers and Scalar
Buffer a = Buffer::fromValues<float>(a_vec, Access::ReadOnly);
Buffer b = Buffer::fromValues<float>(b_vec, Access::ReadOnly);
Buffer c = Buffer::empty<float>(4096, Access::ReadWrite);

// KernelExecutor - program flow
KernelExecutor executor(program);
executor.addKernelCall("vector_add_f32", {&a, &b, &c});

// Running
workingGroup.runOnOneDevice(executor);

// Result
vector<float> result = c.readBack<float>(program.getDevices()[0]->getCommandQueue());
Utils::writeVectorToFile<float>("vector_add_f32", "c.txt", result);
```

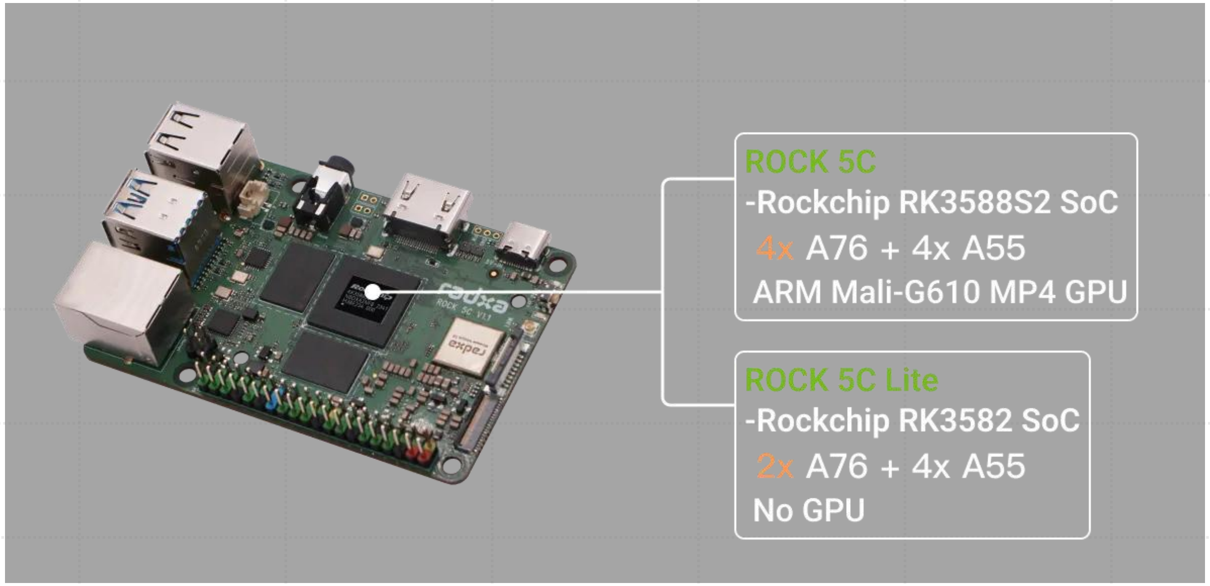
LOC fără a folosi framework-ul: 134

Implementarea directă în OpenCL necesită aproximativ de 6 ori mai mult cod decât utilizarea framework-ului.

```cpp
// Device selection
cl_uint numPlatforms = 0;
clGetPlatformIDs(0, nullptr, &numPlatforms);
vector<cl_platform_id> platforms(numPlatforms);
clGetPlatformIDs(numPlatforms, platforms.data(), nullptr);

vector<cl_device_id> allDevices;
vector<string> deviceNames;

for (auto platform : platforms) {
    cl_uint numDevices = 0;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, nullptr, &numDevices);
    vector<cl_device_id> devices(numDevices);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, numDevices, devices.data(), nullptr);

    for (auto device : devices) {
        size_t nameSize = 0;
        clGetDeviceInfo(device, CL_DEVICE_NAME, 0, nullptr, &nameSize);
        vector<char> nameBuffer(nameSize);
        clGetDeviceInfo(device, CL_DEVICE_NAME, nameSize, nameBuffer.data(), nullptr);
        string deviceName(nameBuffer.data());

        deviceNames.push_back(deviceName);
        allDevices.push_back(device);
    }
}

cout << "Choose a device:\n";
for (int i = 0; i < deviceNames.size(); ++i)
    cout << "[" << i << "] " << deviceNames[i] << "\n";

int selection = 0;
cout << "Index: ";
cin >> selection;

cl_device_id selectedDevice = allDevices[selection];
cl_int err;
cl_context context = clCreateContext(nullptr, 1, &selectedDevice, nullptr, nullptr, &err);
if (!context || err != CL_SUCCESS)
    throw runtime_error("Failed to create OpenCL context.");

cl_command_queue queue = clCreateCommandQueue(context, selectedDevice, CL_QUEUE_PROFILING_ENABLE, &err);
if (!queue || err != CL_SUCCESS)
    throw runtime_error("Failed to create command queue.");

// KernelFiles
ifstream sourceFile("kernels/vector_add.cl");
if (!sourceFile.is_open())
    throw runtime_error("Could not open vector_add.cl file.");

stringstream sourceBuffer;
sourceBuffer << sourceFile.rdbuf();
string sourceCode = sourceBuffer.str();
const char* sourceStr = sourceCode.c_str();

// Host memory initialization
vector<float> a_vec, b_vec;
{
    ifstream in_a("data/vector_add_f32/a.txt");
    throw runtime_error("Could not open a.txt");
    float val;
    while (in_a >> val) a_vec.push_back(val);
}

{
    ifstream in_b("data/vector_add_f32/b.txt");
    throw runtime_error("Could not open b.txt");
    float val;
    while (in_b >> val) b_vec.push_back(val);
}

// KernelFunctions
const char* kernelName = "vector_add_f32";
size_t globalSize = 4096;
```

```cpp
// Program initialization
cl_program program = clCreateProgramWithSource(context, 1, &sourceStr, nullptr, nullptr);
err = clBuildProgram(program, 1, &selectedDevice, nullptr, nullptr, nullptr);
if (err != CL_SUCCESS) {
    size_t logSize;
    clGetProgramBuildInfo(program, selectedDevice, CL_PROGRAM_BUILD_LOG, 0, nullptr, &logSize);
    vector<char> log(logSize);
    clGetProgramBuildInfo(program, selectedDevice, CL_PROGRAM_BUILD_LOG, logSize, log.data(), nullptr);
    cerr << "Build error:\n" << log.data() << '\n';
    throw runtime_error("OpenCL build failed.");
}

// OpenCL Data - Buffers and Scalar
cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * a_vec.size(), nullptr, nullptr);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * b_vec.size(), nullptr, nullptr);
cl_mem bufC = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * 4096, nullptr, nullptr);

cl_event writeEventA, writeEventB;
clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0, sizeof(float) * a_vec.size(), a_vec.data(), 0, nullptr, &writeEventA);
clEnqueueWriteBuffer(queue, bufB, CL_TRUE, 0, sizeof(float) * b_vec.size(), b_vec.data(), 0, nullptr, &writeEventB);

cl_ulong start, end;
double writeTimeA, writeTimeB;

clGetEventProfilingInfo(writeEventA, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, nullptr);
clGetEventProfilingInfo(writeEventA, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, nullptr);
writeTimeA = (end - start) * 1e-6;

clGetEventProfilingInfo(writeEventB, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, nullptr);
clGetEventProfilingInfo(writeEventB, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, nullptr);
writeTimeB = (end - start) * 1e-6;

cout << "Write time A: " << writeTimeA << " ms\n";
cout << "Write time B: " << writeTimeB << " ms\n";
// KernelExecutor - program flow
cl_kernel kernel = clCreateKernel(program, kernelName, nullptr);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);

// Running
double kernelTime, readTime;

cl_event execEvent;
clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &globalSize, nullptr, 0, nullptr, &execEvent);
clFinish(queue);

clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, nullptr);
clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, nullptr);
kernelTime = (end - start) * 1e-6;

// Result
vector<float> result(4096);
cl_event readEvent;
clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0, sizeof(float) * 4096, result.data(), 0, nullptr, &readEvent);
clFinish(queue);

clGetEventProfilingInfo(readEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, nullptr);
clGetEventProfilingInfo(readEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, nullptr);
readTime = (end - start) * 1e-6;

cout << "Kernel execution time: " << kernelTime << " ms\n";
cout << "Readback time: " << readTime << " ms\n";

ofstream out("data/vector_add_f32/c.txt");
for (float val : result)
    out << val << '\n';
```

Platformele folosite pentru testarea framework-ului

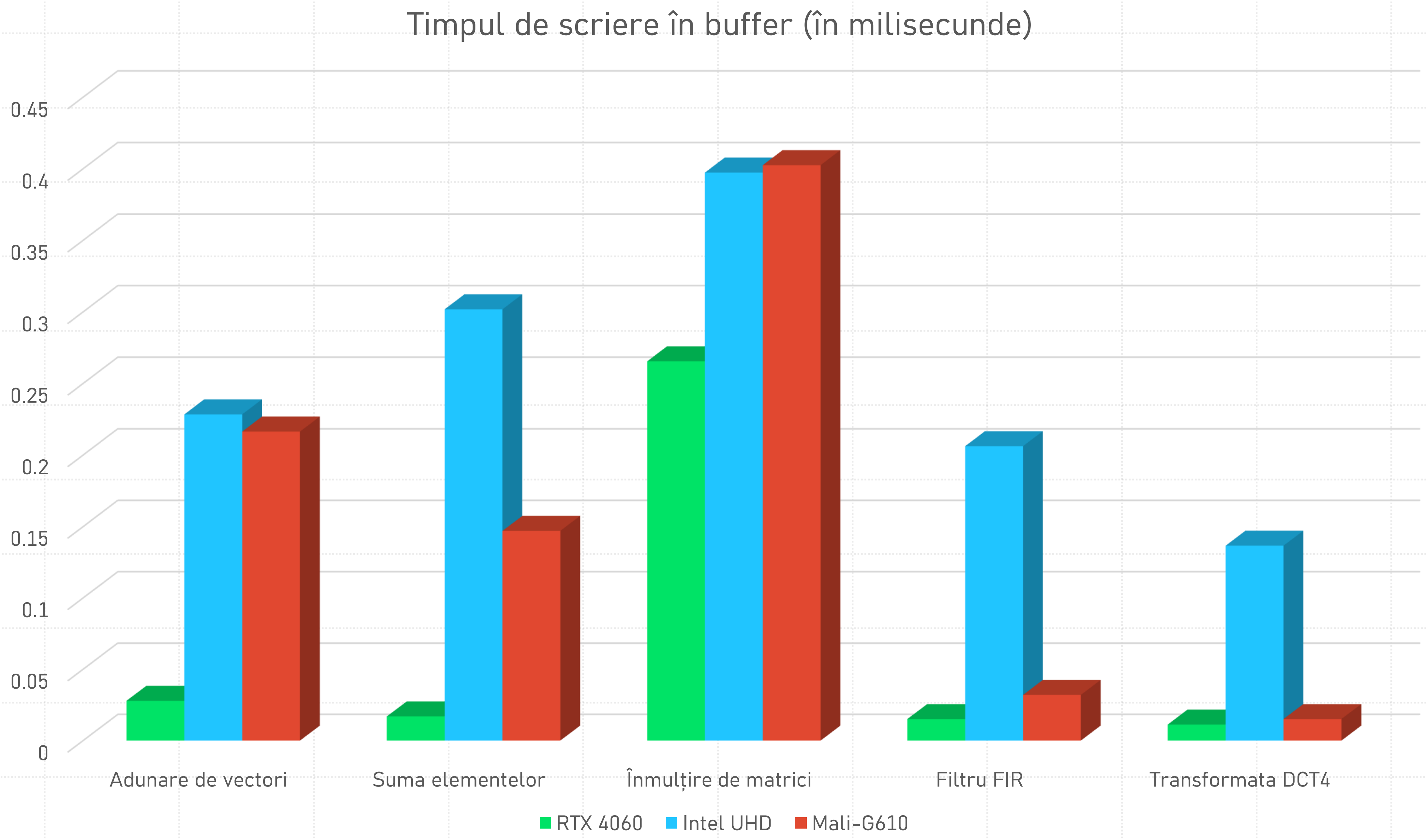


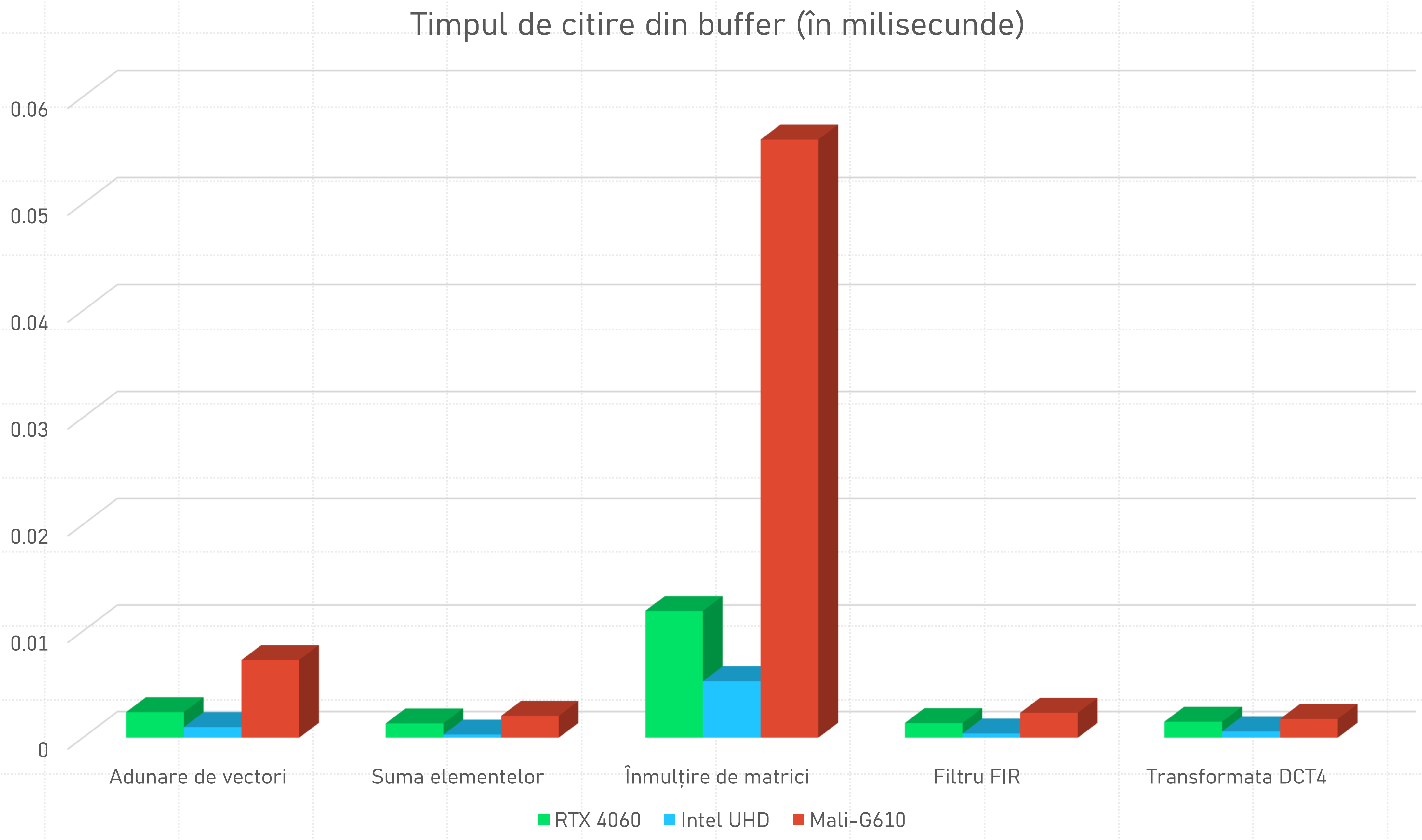
Algoritmii folosiți pentru testarea performanței

- Adunare de vectori (2 vectori cu 4096 de elemente)
- Suma elementelor dintr-un vector (4096 de elemente)
- Înmulțire de matrici (Matrice 1: 256x512, Matrice 2: 512x128)
- Filtru FIR (Bloc de intrare cu 384 de valori și 256 coeficienți) *
- Transformata DCT4 (Vector de dimensiune 2048) *

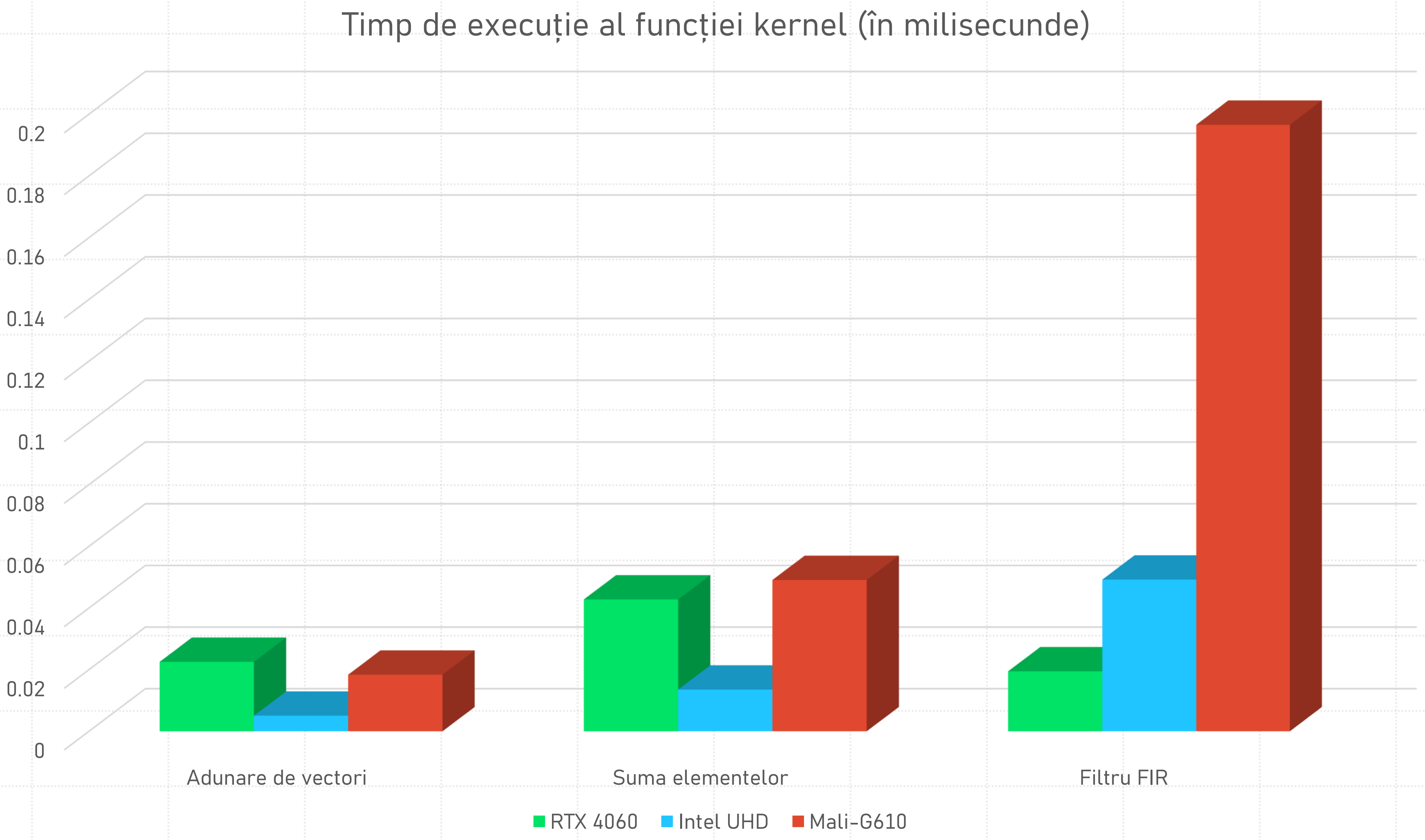
Toate elementele acestor vectori și matrici sunt float32.

*Acești algoritmi au fost luați din articolul: “Embench IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks”

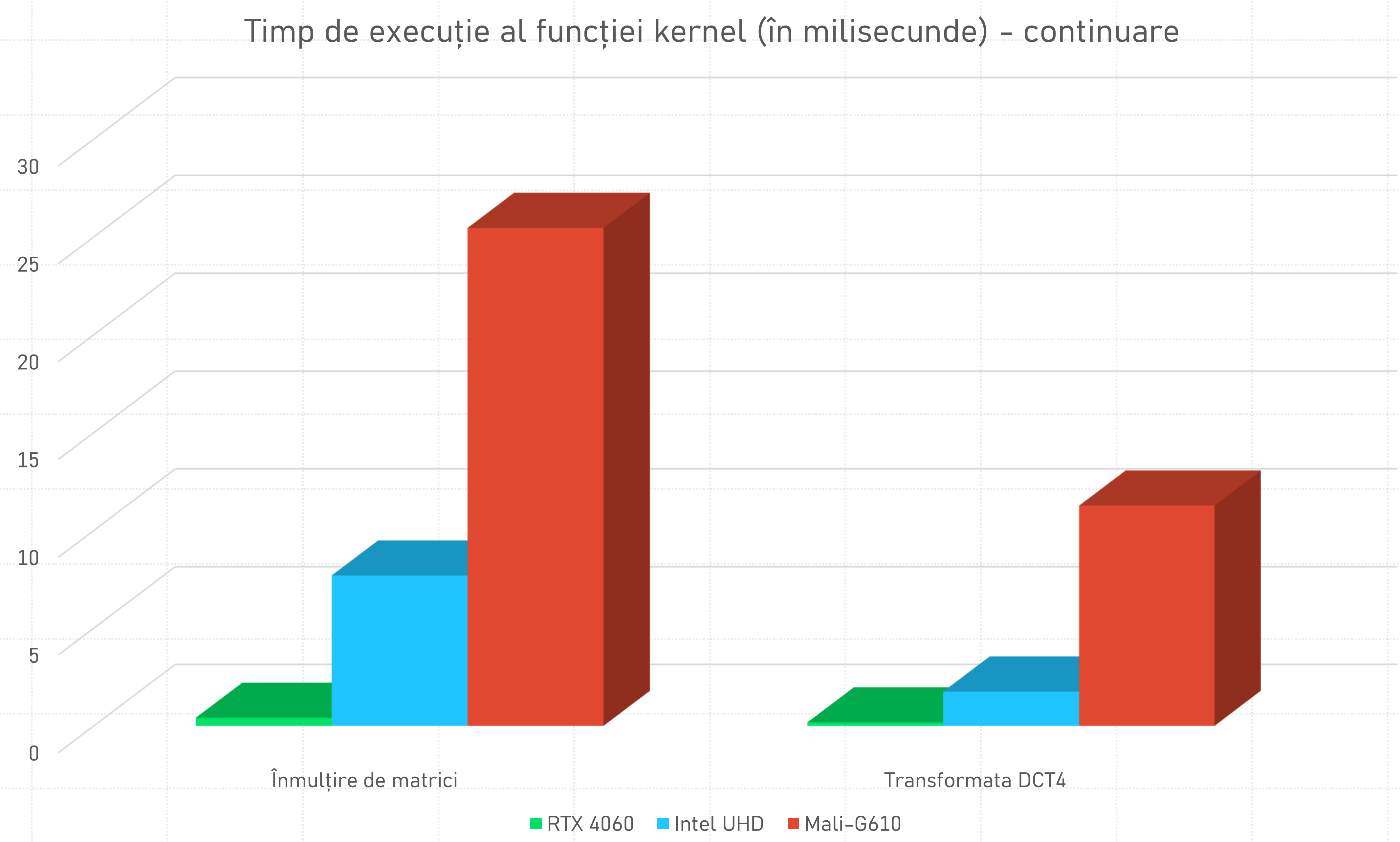

Timpul de scriere în buffer (în milisecunde)
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Adunare de vectori
Suma elementelor
Înmulțire de matrici
Filtru FIR
Transformata DCT4
RTX 4060
Intel UHD
Mali-G610



Timpul de citire din buffer (în milisecunde)
0.06
0.05
0.04
0.03
0.02
0.01
0
Adunare de vectori
Suma elementelor
Înmulțire de matrici
Filtru FIR
Transformata DCT4
RTX 4060
Intel UHD
Mali-G610



Timp de execuție al funcției kernel (în milisecunde)
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
Adunare de vectori
Suma elementelor
Filtru FIR
RTX 4060
Intel UHD
Mali-G610


Timp de execuție al funcției kernel (în milisecunde) - continuare
30
25
20
15
10
5
0
Înmulțire de matrici
Transformata DCT4
RTX 4060
Intel UHD
Mali-G610

Concluzii și direcții viitoare

Concluzii

- Framework-ul simplifică dezvoltarea și testarea aplicațiilor OpenCL
- Este modular, portabil și compatibil cu mai multe tipuri de dispozitive
- Suportă execuția de funcții kernel înlănțuite

Direcții viitoare de dezvoltare

- Execuția distribuită automată pe mai multe dispozitive
- Mecanism de fallback
- Interfață grafică mai amplă

Întrebări?