Programming the GPU in Java
A Little Background
Running Programs on the GPU
Advent of the GPGPU
OpenCL and Java
CUDA and Java
Staying Above Low-Level Code
Conclusion

CODING

Programming the GPU in Java

Accessing the GPU from Java unleashes remarkable firepower. Here's how the GPU works and how to access it from Java.

by Dmitry Aleksandrov

January 10, 2020

Programming a graphics processing unit (GPU) seems like a distant world from Java programming. This is understandable, because most of the use cases for Java are not applicable to GPUs. Nonetheless, GPUs offer teraflops of performance, so let's explore their possibilities.

To make the topic approachable, I'll spend some time explaining GPU architecture along with a little history, which will make it easier to dive into programming the hardware. Once I've shown how the GPU differs from CPU computing, I'll show how to use GPUs in the Java world. Finally, I will describe the leading frameworks and libraries available for writing Java code and running it on GPUs, and I'll provide some code samples.

A Little Background

The GPU was first popularized by Nvidia in 1999. It is a special processor designed to process graphical data before it is transferred to the display. In most cases, it enables some of the computation to be offloaded from the CPU, thus freeing CPU resources while speeding up those offloaded computations. The result is that more input data can be processed and presented at much higher output resolutions, making the visual representation more attractive and the frame rate more fluid.

The nature of 2D/3D processing is mostly matrix manipulation, so it can be handled with a massively parallel approach. What would be an effective approach for image processing? To answer this, let's compare the architecture of standard CPUs (shown in **Figure 1**) and GPUs.

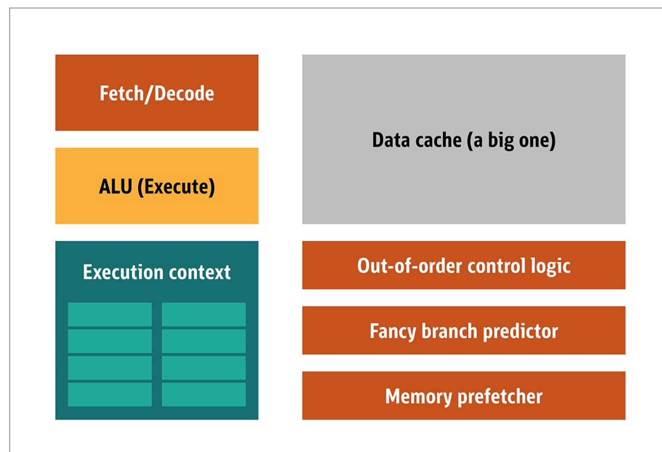

Figure 1. Block architecture of a CPU

In the CPU, the actual processing elements—the fetchers, the arithmetic logic unit (ALU), and the execution contexts—are just a small part of the whole system. To speed up the irregular calculations arriving in

unpredictable order, there are a large, fast, and expensive cache; different kinds of prefetchers; and branch predictors.

You don't need all of this on a GPU, because the data is received in a predictable manner and the GPU performs a very limited set of operations on the data. Thus, it is possible to make a very small and inexpensive processor with a block architecture similar to that in **Figure 2**.

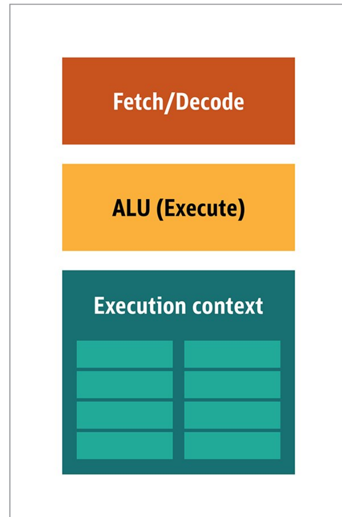

Figure 2. Block architecture for a simple GPU core

Because these kinds of processors are cheap and they process data in parallel chunks, it's easy to put many of them to work in parallel. This design is referred to as *multiple instruction, multiple data* or MIMD (pronounced "mim-dee").

A second approach focuses on the fact that often a single instruction is applied to multiple data items. This is known as *single instruction, multiple data* or SIMD (pronounced "sim-dee"). In this design, a single GPU contains multiple ALUs and execution contexts, with a small area dedicated to shared context data, as shown in **Figure 3**.

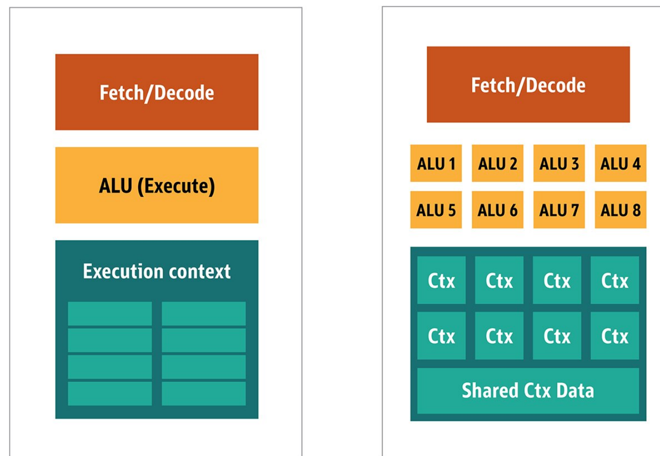

Figure 3. Comparing a MIMD-style GPU block architecture (left) with a SIMD design (right)

Combining SIMD and MIMD processing provides the maximal processing throughput, which I'll discuss shortly. In such a design, you have multiple SIMD processors running in parallel, as shown in **Figure 4**.

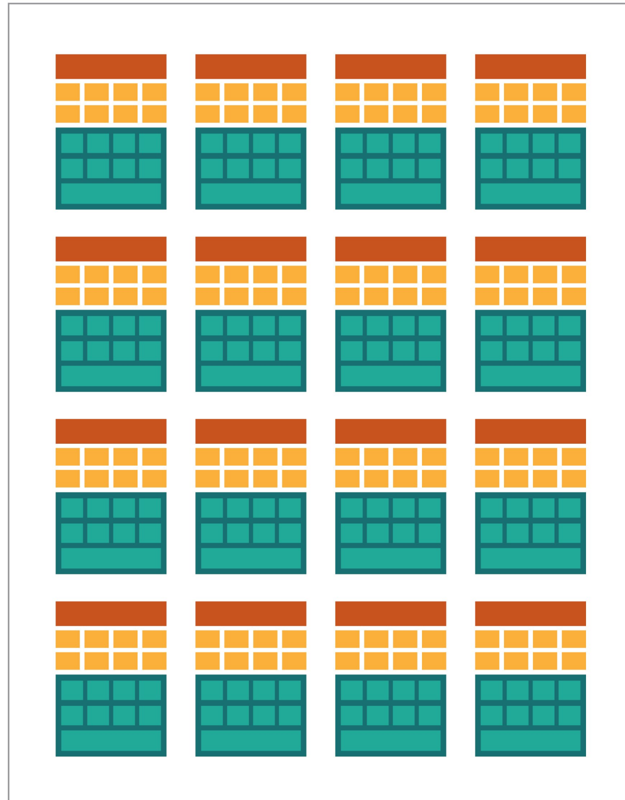

Figure 4. Running multiple SIMD processors in parallel; here, 16 cores with 128 total ALUs

Because you have a bunch of small, simple processors, you can program them to gain special effects in the output.

Running Programs on the GPU

Most of the early visual effects in games were actually hardcoded small programs running on a GPU and applied to the data stream from the CPU.

It was obvious, even then, that hardcoded algorithms were insufficient, especially in game design, where visual representation is actually one of the main selling points. In response, the big vendors opened access to GPUs, and then third-party developers could code for them.

The typical approach was to write small programs, called *shaders*, in a special language (usually a subset of C) and compile them with a special compiler for the corresponding architecture. The term *shaders* was chosen because shaders were often used to control lighting and shading effects, but there's no reason they can't handle other special effects.

Each GPU vendor had its own specific programming language and infrastructure for creating shaders for its hardware. From these efforts, several platforms have been created. The major ones include

- DirectCompute: A proprietary shader language/API from Microsoft that is part of Direct3D, starting with DirectX 10
- AMD FireStream: An ATI/Radeon proprietary technology, which was discontinued by AMD
- OpenACC: A multivendor-consortium parallel computing solution
- C++ AMP: A Microsoft proprietary library for data parallelism in C++
- CUDA: Nvidia's proprietary platform, which uses a subset of the C language
- OpenCL: A common standard originally designed by Apple but now managed by the consortium Khronos Group

Most of the time, working with GPUs is low-level programming. To make it a little bit more understandable for developers to code, several abstractions are provided. The most famous are DirectX, from Microsoft, and OpenGL, from the Khronos Group. These APIs are for writing high-level code, which then can be offloaded to the GPU mostly seamlessly by the developer.

As far as I know, there is no Java infrastructure that supports DirectX, but there is a nice binding for OpenGL. JSR 231 was started in 2002 to address GPU programming, but it was abandoned in 2008 and supported only OpenGL 2.0. Support of OpenGL has been continued in an independent project called JOCL, (which also supports OpenCL), and it's publicly available. By the way, the famous Minecraft game was written with JOCL underneath.

Advent of the GPGPU

Still, Java and GPUs are not a seamless fit, although they should be. Java is heavily used in enterprises, data science, and the financial sector, where many computations and a lot of processing power are needed. This is how the idea of the general-purpose GPU (GPGPU) came about.

The idea to use the GPU this way started when the vendors of video adapters started to open the frame buffer programmatically, enabling developers to read the contents. Some hackers recognized that they could then use the full power of the GPU for general-purpose computations. The recipe was straightforward:

1. Encode the data as a bitmap array.
2. Write a shader to process it.
3. Submit them both to the video card.
4. Retrieve the result from the frame buffer.
5. Decode the data from the bitmap array.

This is a very simplified explanation. I'm not sure this process was ever heavily used in production, but it did work.

Then several researchers from Stanford University began looking for a way to make using a GPGPU easier. In 2005 they released BrookGPU, which was a small ecosystem that included a language, a compiler, and a runtime.

BrookGPU compiled programs written in the Brook stream programming language, which is a variant of ANSI C. It could target OpenGL v1.3+, DirectX v9+, or AMD's Close to Metal for the computational back end, and it ran on both Microsoft Windows and Linux. For debugging, BrookGPU could also simulate a virtual graphics card on the CPU.

However, it did not take off, because of the hardware available at the time. In the GPGPU world, you need to copy the data to the device (in this context, *device* refers to the GPU and the board on which it is situated), wait for the GPU to process the data, and then copy the data back to the main runtime. This creates a lot of latency. And in the mid-2000s, when the project was under active development, this latency almost precluded extensive use of GPUs for general computing.

Nevertheless, many companies saw a future in this technology. Several video adapter vendors started providing GPGPUs with their proprietary technologies, and others formed alliances to provide more-general, versatile programming models to run a larger variety of hardware devices.

Now that I've shared this background, let's examine the two most successful technologies for GPU computing—OpenCL and CUDA—and see how Java works with them.

OpenCL and Java

Like many other infrastructure packages, OpenCL provides a base implementation in C. It is technically accessible via Java Native Interface (JNI) or Java Native Access (JNA), but such access would be a bit too much work for most developers. Fortunately, this work has already been done by several libraries: JOCL, JogAmp, and JavaCL. Unfortunately, JavaCL is a dead project. But the JOCL project is alive and quite up to date. I will use it in the following examples.

But first I should explain what OpenCL is. As I mentioned earlier, OpenCL provides a very general model, suitable for programming all sorts of devices—not only GPUs and CPUs but even digital signal processors (DSPs) and field-programmable gate arrays (FPGAs) as well.

Let's explore the easiest example: vector addition, probably the most representative and simple example. You have two integer arrays you're adding and one resulting array. You take an element from the first array and an element from the second array, and then you put the sum of them in the result array, as shown in **Figure 5**.

```
[ 5, 6, 9, 5, 1, 8, 4..>
  +  +  +  +  +  +  +
[ 1, 2, 0, 1, 5, 1, 5..>
  =  =  =  =  =  =  =
[ 6, 8, 9, 6, 6, 9, 9..>
```

Figure 5. Adding the contents of two arrays and storing the sums in a result array

As you can see, the operation is highly concurrent and thus very parallelizable. You can push each of the add operations to a separate GPU core. This means that if you have 2,048 cores, as on an Nvidia 1080 graphics card, you can perform 2,048 simultaneous add operations. That means there are potentially teraflops of computing power waiting for you. Here is the code for arrays with 10 million integers taken from the JOCL site:

```
public class ArrayGPU {
    /**
     * The source code of the OpenCL program
     */
    private static String programSource =
        "__kernel void "+
        "sampleKernel(__global const float *a,"+
        "             __global const float *b,"+
        "             __global float *c)"+
        "{"+
        "    int gid = get_global_id(0);"+
        "    c[gid] = a[gid] + b[gid];"+
        "}";

    public static void main(String args[])
    {
        int n = 10_000_000;
        float srcArrayA[] = new float[n];
        float srcArrayB[] = new float[n];
        float dstArray[] = new float[n];
        for (int i=0; i<n; i++)
        {
            srcArrayA[i] = i;
            srcArrayB[i] = i;
        }
        Pointer srcA = Pointer.to(srcArrayA);
        Pointer srcB = Pointer.to(srcArrayB);
        Pointer dst = Pointer.to(dstArray);


        // The platform, device type and device numl
        // that will be used
        final int platformIndex = 0;
        final long deviceType = CL.CL_DEVICE_TYPE_AI
        final int deviceIndex = 0;

        // Enable exceptions and subsequently omit 
        CL.setExceptionsEnabled(true);

        // Obtain the number of platforms
        int numPlatformsArray[] = new int[1];
        CL.clGetPlatformIDs(0, null, numPlatformsAr
```

```
        int numPlatforms = numPlatformsArray[0];

        // Obtain a platform ID
        cl_platform_id platforms[] = new cl_platforn
        CL.clGetPlatformIDs(platforms.length, platf
        cl_platform_id platform = platforms[platforn

        // Initialize the context properties
        cl_context_properties contextProperties = ne
        contextProperties.addProperty(CL.CL_CONTEXT_

        // Obtain the number of devices for the plat
        int numDevicesArray[] = new int[1];
        CL.clGetDeviceIDs(platform, deviceType, 0, r
        int numDevices = numDevicesArray[0];

        // Obtain a device ID
        cl_device_id devices[] = new cl_device_id[nu
        CL.clGetDeviceIDs(platform, deviceType, numI
        cl_device_id device = devices[deviceIndex];

        // Create a context for the selected device
        cl_context context = CL.clCreateContext(
            contextProperties, 1, new cl_device_id[
            null, null, null);

        // Create a command-queue for the selected c
        cl_command_queue commandQueue =
            CL.clCreateCommandQueue(context, device

        // Allocate the memory objects for the input
        cl_mem memObjects[] = new cl_mem[3];
        memObjects[0] = CL.clCreateBuffer(context,
            CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOS
            Sizeof.cl_float * n, srcA, null);
        memObjects[1] = CL.clCreateBuffer(context,
            CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOS
            Sizeof.cl_float * n, srcB, null);
        memObjects[2] = CL.clCreateBuffer(context,
            CL.CL_MEM_READ_WRITE,
            Sizeof.cl_float * n, null, null);

        // Create the program from the source code
        cl_program program = CL.clCreateProgramWithS
            1, new String[]{ programSource }, null,

        // Build the program
        CL.clBuildProgram(program, 0, null, null, nu

        // Create the kernel
        cl_kernel kernel = CL.clCreateKernel(progran

        // Set the arguments for the kernel
        CL.clSetKernelArg(kernel, 0,
            Sizeof.cl_mem, Pointer.to(memObjects[0]
        CL.clSetKernelArg(kernel, 1,
            Sizeof.cl_mem, Pointer.to(memObjects[1]
        CL.clSetKernelArg(kernel, 2,
            Sizeof.cl_mem, Pointer.to(memObjects[2]

        // Set the work-item dimensions
        long global_work_size[] = new long[]{n};
        long local_work_size[] = new long[]{1};

        // Execute the kernel
        CL.clEnqueueNDRangeKernel(commandQueue, kerr
            global_work_size, local_work_size, 0, nu

        // Read the output data
        CL.clEnqueueReadBuffer(commandQueue, memObje
            n * Sizeof.cl_float, dst, 0, null, null

        // Release kernel, program, and memory objec
        CL.clReleaseMemObject(memObjects[0]);
        CL.clReleaseMemObject(memObjects[1]);
        CL.clReleaseMemObject(memObjects[2]);
        CL.clReleaseKernel(kernel);
        CL.clReleaseProgram(program);
        CL.clReleaseCommandQueue(commandQueue);
        CL.clReleaseContext(context);

    }

    private static String getString(cl_device_id dev
```

```
            // Obtain the length of the string that wil
            long size[] = new long[1];
            CL.clGetDeviceInfo(device, paramName, 0, nu

            // Create a buffer of the appropriate size
            byte buffer[] = new byte[(int)size[0]];
            CL.clGetDeviceInfo(device, paramName, buffe

            // Create a string from the buffer (excludi
            return new String(buffer, 0, buffer.length-
        }
    }
```

This code doesn't look like Java, but it actually is. I'll explain the code next; don't spend a lot of time on it now, because I will shortly discuss less complicated solutions.

The code is well documented, but let's do a small walk-through. As you can see, the code is very C-like. This is quite normal, because JOCL is just the binding to OpenCL. At the start, there is some code inside a string, and this code is actually the most important part: It gets compiled by OpenCL and then sent to the video card and executed there. This code is called a *kernel*. Do not confuse this term with an OS kernel; this is the device code. This kernel code is written in a subset of C.

After the kernel comes the Java binding code to set up and orchestrate the device, to chunk the data, and to create proper memory buffers on the device where the data is going to be stored as well as the memory buffers for the resulting data.

To summarize: There is "host code," which is usually a language binding (in this case, Java), and the "device code." You always distinguish what runs on the host and what should run on the device, because the host controls the device.

The preceding code should be viewed as the GPU equivalent of "Hello World!" As you see, the amount of ceremony is vast.

Let's not forget the SIMD capabilities. If your hardware supports SIMD extensions, you can make arithmetic code run much faster. For example, let's look at the matrix multiplication kernel code. This is the code in the raw string of your Java application.

```
__kernel void MatrixMul_kernel_basic(int dim,
                  __global float *A,
                  __global float *B,
                  __global float *C){

    int iCol = get_global_id(0);
    int iRow = get_global_id(1);
    float result = 0.0;
    for(int i=0; i< dim; ++i)
    {
          result +=
          A[iRow*dim + i]*B[i*dim + iCol];
    }
    C[iRow*dim + iCol] = result;
}
```

Technically, this code will work on a chunk of data that was set up for you by the OpenCL framework, with the instructions you supply in the preparation ceremony.

If your video card supports SIMD instructions and is able to process vectors of four floats, a small optimization may turn the previous code into the following code:

```
#define VECTOR_SIZE 4
__kernel void MatrixMul_kernel_basic_vector4(
    size_t dim, // dimension is in single floats
    const float4 *A,
    const float4 *B,
    float4 *C)
```

```
{
    size_t globalIdx = get_global_id(0);
    size_t globalIdy = get_global_id(1);
    float4 resultVec = (float4){ 0, 0, 0, 0 };
    size_t dimVec = dim / 4;
    for(size_t i = 0; i < dimVec; ++i) {
        float4 Avector = A[dimVec * globalIdy + i];
        float4 Bvector[4];
        Bvector[0] = B[dimVec * (i * 4 + 0) + globa
        Bvector[1] = B[dimVec * (i * 4 + 1) + globa
        Bvector[2] = B[dimVec * (i * 4 + 2) + globa
        Bvector[3] = B[dimVec * (i * 4 + 3) + globa
        resultVec += Avector[0] * Bvector[0];
        resultVec += Avector[1] * Bvector[1];
        resultVec += Avector[2] * Bvector[2];
        resultVec += Avector[3] * Bvector[3];
    }

    C[dimVec * globalIdy + globalIdx] = resultVec;
}
```

With this code, you can double the performance.

Cool. You have unlocked the GPU for the Java world! But as a Java developer, do you really want to do all of this binding, write C code, and work with such low-level details? I certainly don't. But now that you have some knowledge of how the GPU architecture is used, let's look at other solutions beyond the JOCL code I've just presented.

CUDA and Java

CUDA is Nvidia's solution to these coding issues. CUDA provides many more ready-to-use libraries for standard GPU operations, such as matrices, histograms, and even deep neural networks. The emerging library list already contains many useful bindings. These are from the JCuda project:

- JCublas: all about matrices
- JCufft: fast Fourier transforms
- JCurand: all about random numbers
- JCusparse: sparse matrices
- JCusolver: factorization
- JNvgraph: all about graphs
- JCudpp: CUDA Data Parallel Primitives Library and some sorting algorithms
- JNpp: image processing on a GPU
- JCudnn: a deep neural network library

I'll describe using JCurand, which generates random numbers. You can directly use it from Java code with no other specific kernel languages. For example:

```
...
int n = 100;
curandGenerator generator = new curandGenerator();
float hostData[] = new float[n];
Pointer deviceData = new Pointer();
cudaMalloc(deviceData, n * Sizeof.FLOAT);
curandCreateGenerator(generator, CURAND_RNG_PSEUDO_I
curandSetPseudoRandomGeneratorSeed(generator, 1234)
curandGenerateUniform(generator, deviceData, n);
cudaMemcpy(Pointer.to(hostData), deviceData,
        n * Sizeof.FLOAT, cudaMemcpyDeviceToHost);
System.out.println(Arrays.toString(hostData));
curandDestroyGenerator(generator);
cudaFree(deviceData);
...
```

Here the GPU is used to create more random numbers of high quality, based on some very strong mathematics.

In JCuda you can also write generic CUDA code and call it from Java by just adding some JAR files to your classpath. See the JCuda documentation for more examples.

Staying Above Low-Level Code

This all looks great, but there is too much ceremony, too much setup, and too many different languages to get this running. Is there a way to use a GPU at least partially?

What if you don't want to think about all of this OpenCL, CUDA, and other internal stuff? What if you just want to code in Java and not think about the internals? The Aparapi project can help. *Aparapi* stands for "a parallel API." I think of it as a kind of Hibernate for GPU programming that uses OpenCL under the hood. Let's look at an example of vector addition.

```
public static void main(String[] _args) {
    final int size = 512;
    final float[] a = new float[size];
    final float[] b = new float[size];

    /* fill the arrays with random values */
    for (int i = 0; i < size; i++){
        a[i] = (float) (Math.random() * 100);
        b[i] = (float) (Math.random() * 100);
    }
    final float[] sum = new float[size];

    Kernel kernel = new Kernel(){
        @Override public void run() {
I           int gid = getGlobalId();
            sum[gid] = a[gid] + b[gid];
        }
    };

    kernel.execute(Range.create(size));
    for(int i = 0; i < size; i++) {
        System.out.printf("%6.2f + %6.2f = %8.2f\n"
    }
    kernel.dispose();
}
```

This is pure Java code (taken from the Aparapi documentation), although here and there, you can spot some GPU domain-specific terms such as `Kernel` and `getGlobalId`. You still need to understand how the GPU is programmed, but you can approach GPGPU in a more Java-friendly way. Moreover, Aparapi provides an easy way to bind OpenGL contexts to the OpenCL layer underneath—thus enabling the data to stay entirely on the video card—and thereby avoid memory latency issues.

If many independent computations need to be done, consider Aparapi. This rich set of examples gives some use cases that are perfect for massive parallel computations.

In addition, there are several projects such as TornadoVM that automatically offload suitable calculations from the CPU to the GPU, thus enabling massive optimizations out of the box.

Conclusion

Although there are many applications where GPUs can bring some game-changing benefits, you might say there are still some obstacles. However, Java and GPUs can do great things together. In this article, I have only scratched the surface of this vast topic. My intention was to show various high- and low-level options for accessing a GPU from Java. Exploring this area will deliver huge performance benefits, especially for

complex problems that require numerous calculations that can be performed in parallel.

Dmitry Aleksandrov

Dmitry Aleksandrov (@bercut2000) is a chief architect at T-Systems, a Java Champion, Oracle Groundbreaker, and blogger. He has more than a decade experience mainly in Java Enterprise in banking/telecom, but he is also interested in dynamic languages on JVM and features such as massive-parallel computations on GPUs. He is a colead of the Bulgarian Java User Group and co-organizer of jPrime Conf. Dmitry is also a frequent speaker at local events as well as conferences such as JavaOne/Code One, Devoxx, JavaZone, and Joker/JPoint.

Share this Page