<http://javagl.de/jcuda.org/tutorial/TutorialIndex.html>

# Getting started with JCuda

## Overview

* General information about JCuda
* The basic setup for JCuda
* The setup of a minimum JCuda test project
* How to create, compile and run CUDA kernels with JCuda

## Futher information:

A detailed article about [GPU Computing Using CUDA, Eclipse, and Java with JCuda](http://www.codeproject.com/Articles/513265/GPU-Computing-Using-CUDA-Eclipse-and-Java-with-JCu) has been published by Mark Bishop. It is an excellent resource for further information about the setup of CUDA and JCuda on Linux platforms, and the setup of a JCuda project in Eclipse.

## Introduction

CUDA provides two different APIs: The Runtime API and the Driver API. Both APIs are very similar concerning basic tasks like memory handling. In fact, starting with CUDA 3.0, both APIs are interoperable and can be mixed to some extent. However, there are some important differences. The most important difference between the Runtime API and the Driver API for JCuda is the way how kernels are managed and executed:

In the original CUDA Runtime API, kernels are defined and compiled together with C files. The source code is compiled by the NVCC, the NVIDIA CUDA Compiler. This compiler uses another C compiler (for example, the GCC or Visual Studio Compiler) to compile the plain C parts of the source code, and takes care of the compilation of the CUDA specific parts, like the CUDA kernels and the kernel<<<...>>> calls. The result of this compilation is usually an executable file comprising the whole program.

Of course, the NVCC can not be used to compile a Java program. The kernel<<<...>>> call syntax can not be used in Java, and there is not a single, executable file after the compilation. Thus, it is not possible to call own CUDA kernels with the JCuda Runtime API. Instead, the JCuda Driver API has to be used, as explained in the section about [Creating kernels](http://javagl.de/jcuda.org/tutorial/TutorialIndex.html#CreatingKernels).

The JCuda Runtime API is mainly intended for the interaction with the Java bindings of the CUDA Runtime libraries, like JCublas and JCufft. A Java programmer who only wants to use these libraries and does not want to create own CUDA kernels can use the JCuda Runtime API for the interaction with these libraries. The [Samples](http://javagl.de/jcuda.org/samples/samples.html) section contains basic example programs for each of the available runtime libraries, which may serve as starting points for own JCuda Runtime programs.

## General setup

In order to use JCuda, you need an installation of the CUDA driver and toolkit, which may be obtained from the [NVIDIA CUDA download site](http://developer.nvidia.com/cuda-toolkit-archive). (Note that there may be some delay between the release of a new CUDA version and the release of the matching JCuda version). You should first install the Developer Drivers for your operating system, and then the matching CUDA Toolkit. Please consult also the documentation from the NVIDIA site for the proper setup and installation procedure.

The SDK and code samples are not required to use JCuda, but the code examples may be helpful to get started and to see whether CUDA is working in general.

After CUDA has been properly installed, you may download the JCuda archive for your operating system from the [downloads](http://javagl.de/jcuda.org/downloads/downloads.html) section.

## For JCuda 0.8.0RC and newer versions

The archives contain the main JAR files, as well as JAR files that contain the native libraries (which are .DLL files for Windows, .SO files for Linux and .DYLIB files for MacOS). All required JAR files have to be present in the CLASSPATH.

## For earlier JCuda versions (below 0.8.0RC)

The archives contain the JAR files, and the matching native libraries (which are .DLL files for Windows, .SO files for Linux and .DYLIB files for MacOS). The JAR files have to be present in the CLASSPATH, and the native library files must be located in a path that is visible for Java. In most cases, this should either be a path that is given as a java.library.path for the JVM, or the root directory of the project. (Alternatively, they can also be in a path that is contained in an environment variable like PATH  environment variable on Windows or the LD\_LIBRARY\_PATH environment variable on Linux).

## 基本测试

This section describes how to manually set up a minimum JCuda project from the command line as a basic test. If you are already familiar with using JARs and native libraries from Java, you may probably skip this section and create a new JCuda project directly in your favorite IDE. Otherwise, you may create a first JCuda project by following these steps:

Copy all files from the downloaded JCuda release archive into one directory, and add the following file as "JCudaRuntimeTest.java" in the same directory:

**import**jcuda.\*;  
**import**jcuda.runtime.\*;  
**public class**JCudaRuntimeTest  
{  
    **public static void**main(String args[])  
    {  
        Pointer pointer = **new**Pointer();  
        JCuda.cudaMalloc(pointer, 4);  
        System.out.println("Pointer: "+pointer);  
        JCuda.cudaFree(pointer);  
    }  
}

* Compile the program using the following command on the command line in the project directory (adjusting the version number "0.8.0" according to the version you are using):

|  |  |
| --- | --- |
| On Windows: | **javac -cp ".;jcuda-0.8.0.jar;jcuda-natives-0.8.0-windows-x86\_64.jar" JCudaRuntimeTest.java** |
| On Linux: | **javac -cp ".:jcuda-0.8.0.jar:jcuda-natives-0.8.0-linux-x86\_64.jar" JCudaRuntimeTest.java** |

* This should create the "JCudaRuntimeTest.class" file in the same directory.
* Start the program with the following command:

|  |  |
| --- | --- |
| On Windows: | java -cp ".;jcuda-0.8.0.jar;jcuda-natives-0.8.0-windows-x86\_64.jar" JCudaRuntimeTest |
| On Linux: | java -cp ".:jcuda-0.8.0.jar:jcuda-natives-0.8.0-linux-x86\_64.jar" JCudaRuntimeTest |

* This should print some information about the pointer which was created in the program.

If you encounter any problem during this test, it's most likely an UnsatisfiedLinkError. You may consider opening a thread in the [Forum](https://forum.byte-welt.net/c/byte-welt-projekte-projects/jcuda). including information about the operating system, CUDA version and JCuda version that you are using. (There is a [Forum FAQ Entry about the UnsatisfiedLinkError](https://forum.byte-welt.net/t/jcuda-faq-please-read-before-posting/2542) which may help to solve this problem for the case that you are using an older JCuda version).

## 创建核函数

As described in the [Introduction](http://javagl.de/jcuda.org/tutorial/TutorialIndex.html#Introduction), own CUDA kernels can be launched in JCuda using the Driver API. This section will describe the basic workflow for creating and compiling a simple kernel, and for loading and executing the kernel with JCuda. Most of the information presented here applies equally to CUDA and JCuda, and more detailed information is available, for example, in the CUDA Programming Guide. This section is mainly intended as a quick start, and to point out potential differences between CUDA and JCuda.

The source code for the example described here is available as the [JCudaVectorAdd](http://javagl.de/jcuda.org/samples/samples.html" \l "JCudaVectorAdd) example from the samples section. The sample tries to compile the kernel at runtime, but the general process of manually compiling a kernel is described here.

## 编写核函数

This kernel code is written exactly in the same way as it is done for CUDA. Usually, the kernel code will be located in an individual file. (In the CUDA Runtime API, the kernel function often is part of a larger C file. While it is still possible to have additional C code in the same file as the kernel, this C code will be ignored and not relevant for JCuda).

There is only one important aspect to consider: When the kernel should be executed with the Driver API (regardless of whether it is used in CUDA or JCuda), the kernel function has to be identified and accessed by specifying its name. But when the code is compiled with a C/C++ compiler, the name of the function will be mangled - that means that the function name will internally be modified depending on its signature, and a simple kernel function name, like "add", may be converted to a name like "\_Z3addiPfS\_S\_". While it is still possible to access the function using this name, it is in general much easier more intuitive to declare the kernel function as an

**extern**"C"

|  |
| --- |
|  |
| function. This way, the original name will be preserved. As a example, here is a kernel which performs a simple vector addition:  extern "C" \_\_global\_\_ void add(int n, float \*a, float \*b, float \*sum) {     int i = blockIdx.x \* blockDim.x + threadIdx.x;     if (i<n)     {         sum[i] = a[i] + b[i];     } } |

## 编译核函数

The kernel source code will have to be compiled by the NVCC compiler. This will create a file that can be loaded and executed using the Driver API. There are basically two options how the kernel can be compiled:

* As a **PTX** file, which is a human-readable (but hardly human-understandable) file containing a specific form of "assembler" source code.
* As a **CUBIN** file, which is a "CUDA binary" and contains the compiled code that can directly be loaded and executed by a specific GPU.

While earlier examples from the [Samples](http://javagl.de/jcuda.org/samples/samples.html) section generally used CUBIN files, they have an important drawback: They are specific for the Compute Capability of the GPU. The Compute Capability is a sort of a version number for the hardware, and CUBIN files that have been created for one Compute Capability can not be loaded on a GPU with a different Compute Capability. Thus, upcoming samples will in general prefer the usage of PTX files(倾向使用PTX文件), since they are compiled at runtime for the GPU of the target machine.

A PTX file can be created from a simple, single CUDA source code file with the following command:  
nvcc -ptx JCudaVectorAddKernel.cu -o JCudaVectorAddKernel.ptx

In order to create a valid CUBIN file, it may be necessary to specify the architecture and Compute Capability of the target machine. The full command line for creating a CUBIN file for a GPU with Compute Capability 2.1 on a 64 bit machine would be

**nvcc -cubin -m64 -arch sm\_21 JCudaVectorAddKernel.cu -o JCudaVectorAddKernel.cubin**

For more information about the NVCC and its command line parameters, see the documentation of the NVCC in the /doc/ directory of your CUDA Toolkit installation.

## 在JCuda中加载和执行核函数

|  |
| --- |
| The process of loading and executing a kernel from a PTX- or CUBIN file in the JCuda Driver API is the same as in the CUDA Driver API. The most simple example of a single kernel will be summarized here. First of all, the PTX- or CUBIN file has to be loaded, and a pointer to the kernel function has to be obtained:  // Load the ptx file. CUmodule module = new CUmodule(); cuModuleLoad(module, "JCudaVectorAddKernel.ptx");  // Obtain a function pointer to the kernel function. CUfunction function = new CUfunction(); cuModuleGetFunction(function, module, "add"); |
| Note that the cuModuleLoad function will automatically detect the type of the specified file. So to load a CUBIN file, the same function can be used.  For calling the kernel, some of the language-specific limitations of Java may become more obvious. The functions for setting up the kernel parameters had been rather difficult to use up to CUDA 3.2, and in CUDA 4.0, these functions have been replaced by a single function. This function receives all parameters that describe the kernel execution. Additionally, it receives all kernel parameters in a single void\*\* pointer. A void\*\* pointer is emulated using the Pointer class in JCuda. With this class, the setup of the kernel parameters may even be simpler in JCuda than in CUDA:  // Set up the kernel parameters: A pointer to an array // of pointers which point to the actual values. Pointer kernelParameters = Pointer.to(     Pointer.to(new int[]{numElements}),      Pointer.to(deviceInputA),      Pointer.to(deviceInputB),      Pointer.to(deviceOutput) );  // Call the kernel function. cuLaunchKernel(function,      gridSizeX,  1, 1,       // Grid dimension      blockSizeX, 1, 1,       // Block dimension     0, null,                // Shared memory size and stream      kernelParameters, null // Kernel- and extra parameters ); |

However, one has to take the same care here as in C: The number of pointer indirections has to be verified carefully. Having to create a pointer to a pointer to a pointer in order to pass a pointer as one parameter to a kernel may look confusing at the first glance, but the existing sample programs should help to get this right, and afterwards, the same pattern can be applied to nearly all kernel launches.