2012년 12월 7일

University of Seoul

University Carlos III of Madrid

Computer Engineering

Final project

Evaluating different Java bindings for OpenCL

Raquel Medina Dominguez

Supervisor at University of Seoul: Youngmin Yi ymyi@uos.ac.kr

Supervisor at University Carlos III: Jose Maria Sierra Camara sierra@inf.uc3m.es

University of Seoul 7th December 2012

Acknowledgments

I would like to thanks my supervisor at the University of Seoul, Youngmin Yi, for helping me with his knowledge and for the time he spent with my work, also to my supervisor in the University Carlos III of Madrid, Jose Maria Sierra Camara.

Special thanks to my family and friends. I have no words to describe how grateful I am to my father Jesus, my mother Maria de los Angeles and my sister Maria. I am sure without their support I could not endure all the stress and difficulties arose from all these years of studies.

I want to mention here to my partners in the Parallel System Lab. They have been like a family to me, and they helped me in all they could. Special thanks to Hyun-Woo Lee, Young-Sang Woo and Saehanseul Yi.

I also want to mention my best friends during my period in Seoul, Aileen and Lara, for being so patient with me, for listening to me everyday, and for helping me in every situation. Also mention here the program SeoulMate for introducing to me my buddy Sooin and Hayley and I cannot forget here my friend and personal translator 백여사

Declaration

I hereby declare that the presented final project is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final project in my home university, University Carlos III of Madrid and the University of Seoul.

I acknowledge that my final project is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs in corporates there in or attached there to and all corresponding documentation (here in after collectively referred to as the Work), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non- profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In University of Seoul 7th December, 2012 .

Abstract

The traditional CPU is able to run only a few complex threads concurrently. By contrast, a GPU (Graphics Processing Unit) allows a concurrent execution of hundreds or thousands of simpler threads. The GPU was originally designed for a computer graphics, but nowadays it is being used for general-purpose computation using a GPGPU (General Purpose GPU) technology.

OpenCL, one of the GPGPU technologies, is introduced in this final project. OpenCL is an extension of C and enables efficient parallel programming for heterogeneous devices including both multi-core CPUs and GPUs. However, it provides a low level abstraction to utilize the hardware efficiently. This tends to a hurdle for productive parallel programming. On the other hand, Java is widely used in many application domains since it provides good productivity in software development. Recently, several methods that bind OpenCL and Java have been suggested: Joagamp, Jocl, JavaCL. In this final project, I evaluate these Java bindings for OpenCL in terms of execution time and the memory used. My own class for vector multiplication has been the baseline application in evaluating the libraries presented here. My results show that Joagamp is more efficient, and Jocl consumes less memory, while JavaCL is most productive in terms of the number of lines of code.

Keywords java, opencl, jogamp, javacl, jocl

Abstracto

Por una lado el CPU tradicional es capaz de ejecutar solo varios threads al mismo tiempo. Por otro lado, la tecnologia GPU permite ejecutar cientos o miles de simples threads. La tecnologia GPU fue originalmente disenada para graficos pero en estos dias esta siendo usada para calculos usando GPGPU tecnologia.

OpenCL, una de las GPGPU tecnologias, es introducida en este projecto final. Metodos de coperacion entre Java y el lenguaje presentado. Varias librerias son prsentadas en este proyecto como Jogamp, JOCL y JavaCL.

Mi propia clase multiplicacion de vectores ha sido usada como base de nuestra aplicaion para evaluar las diferentes librerias presentadas aqui.

Diferentes medidas han sido usadas para evaular estas diferentes plataformas como son velocidad, tiempo y memoria usada. Acorde a estas medidas somos capaces de definir que tipo de libreria es mas adecuada para los diferntes proyectos que se deseen elaborar.

Palabras clave java, opencl, jogamp, javacl, jocl

Contents

List of figures

List of tables

1. Introduction

In the recent years, the design of computer architectures has been focused on increasing the thread-level parallelism to increase the performance. The classical approach to improve the performance was to increase the clock frequency. However, that approach has been left behind due to the power consumption problem: Power is proportional to the clock frequency. To resolve this problem, multi-core architectures have emerged. By lowering the frequency (or maintaining the same frequency) and by increasing the number of cores in a CPU, we can achieve more performance while meeting the power consumption constraints.

On the other hand, influenced by the video game industry, GPUs have been evolved into highly parallel computational units, which are programmable and provide high memory bandwidth. Today, the vast majority of computer systems include CPUs, GPUs and other processors. It is necessary to dispose of software that is able to harness the computing power present in these heterogeneous architectures.

OpenCL is a standard computing platform for heterogeneous systems arising in December 2008, which has gained increasing importance. This is mainly due to their efficiency and their compatibility with the vast majority of devices for parallel programming. Additionally, OpenCL is backed by a consortium of companies comprising the Khronos group, among which are companies like NVIDIA, AMD or Apple, prompting further expansion.

However, OpenCL has been developed to be used from C/C++, which limits their use by developers unaccustomed to programming in these languages. On the other hand, the Java programming language is widely used as it provides a higher level of abstraction, allowing better productivity. To bridge the gap between the efficient yet less productive OpenCL programming and the productive yet less efficient Java programming, several Java bindings for OpenCL have been suggested recently. The main contribution of this project is to evaluate different Java bindings for OpenCL in order that a programmer can choose the right binding that works better for her project.

2. Java, OpenCL and Parallel computing overview[1][2]

Lately, consumer architectures are increasingly giving guidance to exploit parallelism to increase performance. It is all about the term Performance Wall . This term refers to the limit reached by processors using traditional techniques of increasing their yield, as the increase on the processor clock frequency. The problem with these techniques appears when you reach the physical limits to increase the benefits. Once the limit is exceeded, it is necessary to use another kind of improvements that will continue to increase the performance of these devices.

For this reason, in recent years the trend has been followed the increase in the clock frequency of a processor, the inclusion of multiple processors, downplaying frequency thereof. The evolution followed by the GPU has been completely different. GPUs were created as parallel computing devices, but are intended to be specific processes such as graphics rendering. However, in recent years, GPU have been evolved into programmable parallel processors, oriented general purpose programming. For this reason there is a need of tools that can exploit the computing power of these heterogeneous platforms.

Create applications for heterogeneous platforms is not simple, because the traditional programming models and those oriented to the development of multi-core platforms and many-core are very different. Traditional models are usually based on standards that assume a common memory space and do not cover explicitly the vector operations.

However, models of general purpose programming on GPU memory add complex hierarchies and vector operations, but are generally dependent hardware, platform and manufacturer.

These limitations make it difficult to access the computing power of the different heterogeneous processors from a single source-code platform. It is also necessary to be aware that in addition to CPU and GPU, heterogeneous architecture can be formed from other devices such as DSP (Digital Signal Processor) or the Cell processor.

Following this purpose, over the past few years, there have been various developer tools that provide the ability to leverage the performance of these new types of processors. These tools give guidance highlights the computation on GPUs, such as CUDA, NVIDIA, and CPUs with multiple processors, such as OpenMP or Ct (Intel). All these tools share the same limitation, be supported only on specific hardware.

Due to increased interest in this area, the need for a cross-platform tool is undeniable .Independent of the manufacturer of the device, and not only be able to take advantage of the graphics device performance, but also of multicore processors and other computing devices, such as embedded processors (eg, DSP). Thus arises OpenCL, a standard for general purpose programming developed by Khronos Group.

Khronos Group is an industry consortium that aims to develop open standards focused on parallel computing and graphics processing on all platforms. These standards are OpenCL, OpenGL, WebGL, The main components of this group are showed in the figure below.

Figure 1:Khronos Group

OpenCL has grown to become the first open source standard for general purpose programming as a heterogeneous architectures, providing efficient access software developers and independent platform for these new devices.

One objective of OpenCL is that all computational resources of the system can be exploited for general purpose computing. OpenCL has been developed for use with C / C + +, a language that provides higher performance, with the penalty of the code under the compilator.

This limitation arises the need to do that OpenCL is, as far as possible, a language all portable devices without code modifications or different builds, but without losing its efficiency. In addition , a developer would be able to develop code that can be compiled into a single machine, but executed in any operating system using different types of devices (CPUs, GPUs, etc..) increasing the original limit of the tool.

Position Nov 2012	Position Nov 2011	Programming Language	Ratings Nov 2012	Delta Nov 2011
1	2	C	19.224%	+1.90%
2	1	Java	17.455%	-0.42%
3	6	Objective-C	10.383%	+4.40%
4	3	C++	9.698%	+1.61%
5	5	PHP	5.732%	-0.36%
6	4	C#	5.591%	-1.73%
7	7	(Visual) Basic	5.032%	-0.01%
8	8	Python	4.062%	+0.45%
9	10	Perl	2.182%	+0.10%
10	11	Ruby	1.739%	+0.24%
11	9	JavaScript	1.278%	-1,29%
12	16	Delphi/Object Pascal	0.995%	+0.12%
13	13	Lisp	0.951%	-0.23%
14	14	Pascal	0.881%	-0.11%
15	23	Visual Basic .NET	0.769%	+0.24%
16	19	Ada	0.662%	+0.04%
17	12	PL/SQL	0.632%	-0.81%
18	18	Lua	0.631%	0.00%
19	15	MATLAB	0.620%	-0.34%
20	24	Assembly	0.585%	0.06

List 1: TIOBE Programming Community Index for November 2012

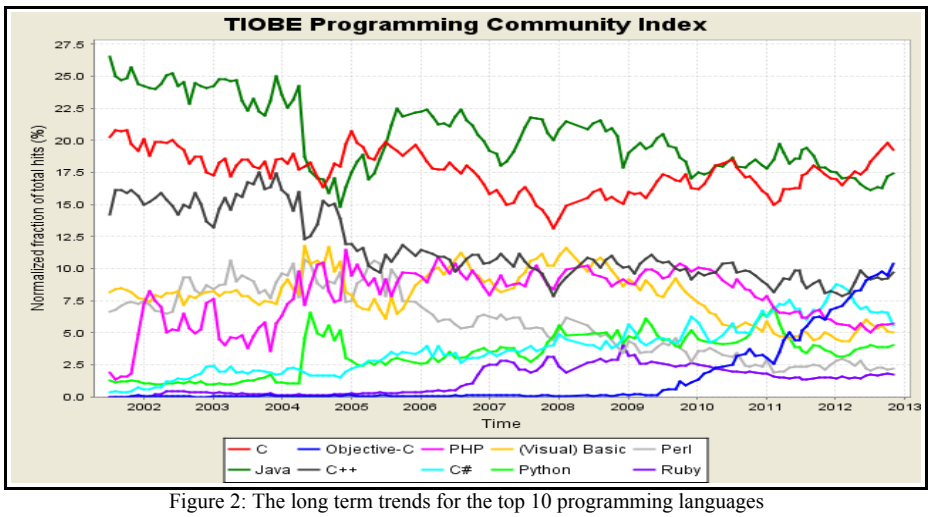

Figure 2: The long term trends for the top 10 programming languages

For this reason, this project presents Java binding to OpenCL, several APIs that gives developers the ability to use OpenCL in Java, one of the most common programming languages.

The union of Java and OpenCL provide all the advantages of the two languages, such as OpenCL efficiency combined with the portability of Java and its treatment of errors. And with Java, it provides the developer with a simpler interface that used-to-date by OpenCL, facilitating learning and subsequent use, and largely avoiding the common problems that may arise inexperienced programmer in C (memory management , pointers, etc..).

2.1 Parallel computing[3]

The speed of conventional sequential computers has continually increased to fit the needs of the application, until to reach the physical limits (Performance Wall).

But in many areas , it still need higher computational power, such as modeling and numerical solution of problems in science and engineering, or costly iterative calculations on large amounts of data with high temporal constraints.

These systems are becoming more complex requiring greater computational capacity. But this is not always possible due to the physical limitations imposed by the development of processors.

To face these limitations , the system have opted for the use of multiple processors forming a parallel system. The parallel system provides a wide range of options to increase performance such as, use of a pipeline, instruction level parallelism, out of order execution and speculation.

Parallel programming is based on the use of multiple processors to solve a common task. The manner in which each processor will face the problem is defined by the developer, so that each processor works on a portion of the problem, exchanging necessary results through shared memory or using a network interconnection.

2.2 Java[4]

Java is a programming language developed by object-oriented Sun Microsystems in the 90s. The syntax is similar to languages like C / C + + programmer but abstracting the low-level tools such as the direct memory access and handling of pointers.

A Java application running on a Java Virtual Machine (JVM), which is responsible for executing the code generated by precompiling the application. This generated code, called bytecode is obtained using the Java compiler, so any virtual machine capable of running it.

The most important aspects of Java are:

1.It is object oriented. This is a programming paradigm that abstracts the data structures used by programmers to objects, entities that are composed of three parts:

• Status: It consists of attributes that store information about the object, which will have specific values

• Behavior: It is defined by the methods that represent the operations that can be performed on objects.

• Identity: This is a property that ensures that each object is different from the rest.

2.Platform-independence. The Java philosophy says that any application written in Java can be run on any plat-form, as its slogan "write once, run everywhere".

3.Garbage collector. In Java you can not release reserved memory objects, this is done by the garbage collector (Garbage Collector - CMS). This collector frees the memory of objects when there are no references to the same signal will not be used in the rest of the code.

2. 3 OpenCL

OpenCL (Open Computing Language) is the name given to the standard of parallel programming architectures developed and released by Khronos. It is backed by major companies that produce hardware and software related to parallel computing, such as AMD, NVIDIA, Apple, IBM, Intel, etc.. This technology is starting to gain importance in the world of general purpose computing on GPUs.

Moreover, being a recognized standard there is no need to learn a programming language on cards for a specific company and another completely different set of cards for a different company. It is only necessary to have compatible drivers and libraries which allow OpenCL development. This has led to a significant increase OpenCL in use since its launch.

Finally, the main point of OpenCL is the portability. It is important to note that this is only functional portability. This is because although the results of the application are correct on different devices, for best performance it is necessary to optimize the code for use in a particular device. For this reason, the same application, still running on two different devices, you will not get the same performance in both.

2.4 GPGPU

GPGPU stands for General-Purpose computation on Graphics Processing Units, which means, general purpose computing on graphics processing units (GPU). GPUs are high performance processors consist of multiple cores capable of conducting major operations on various data with great performance.

Although years ago the GPUs were aimed primarily at graphs and they were very difficult to programming, nowadays ,they have become parallel general purpose processors that support high-level interfaces allowing programming languages like C low / C + +.

The GPU is a suitable platform for the execution of tasks that can be expressed as data parallel computing, which it becomes a very efficient device for problems that may be

parallelism, losing effectiveness against all this sequential problems.

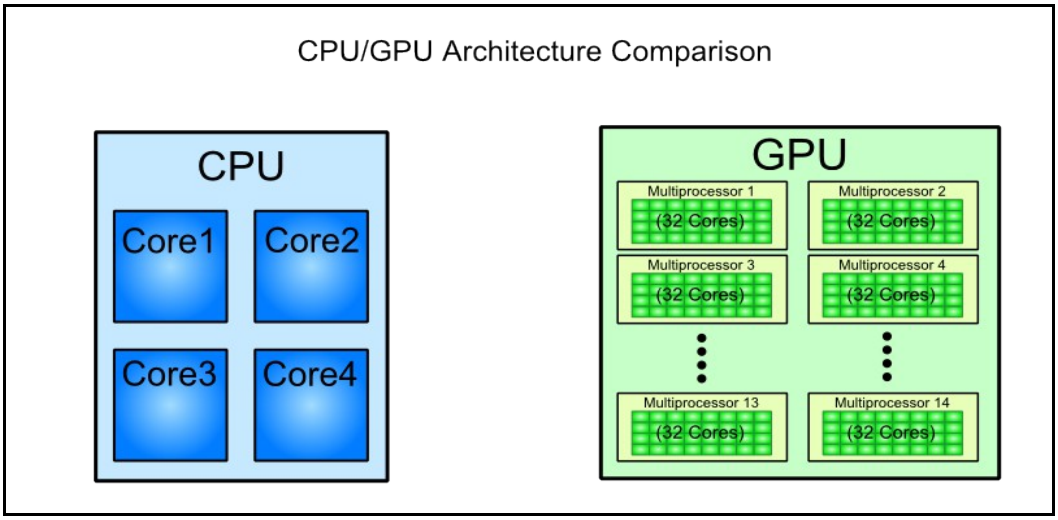

Figure 3: CPU vs GPU architecture [6]

3.Jogamp[7]

3.1Description

Jogamp provides java bindings to OpenCL, the open computing language.

JOCL enables applications running on the JVM to use OpenCL for massively parallel, high performance computing tasks, executed on heterogeneous hardware (GPUs, CPUs, FPGAs etc) in a platform independent manner.

Jogamp is composed of two parts, the low level and the high level binding.

The low level bindings (LLB) are generated from the Khronos OpenCL headers and provide a high performance, JNI based, 1:1 mapping to the C functions.

The advantages of use it can be that reduces maintenance overhead and ensures spec conformance, compile time JNI bindings are fastest way to access native libs from the JVM , makes translation OPENCL C code into Java+JOCL very easy and flexibility and stability(OpenCL libraries are loaded dynamically and accessed via function pointers.

The hand written high level bindings (HLB) is build on top of LLB and hides most boilerplate code (like object IDs, pointers and resource management) behind easy to use java objects. HLB use direct NIO buffers internally for fast memory transfers between the JVM and the OpenCL implementation and is very GC friendly.

The object of this library is to provide an object-oriented abstraction of OpenCL for Java. This simplifies the usage and may be found more natural and accessible for most Java Programmers. The library also offers a low-level interface, which is generated using the GlueGen library. This interface is analogous to the OpenCL API, but not really supposed to be used by clients, and mainly serves as the basis for the object-oriented wrapper.

3.2 How to install it

3.2.1 Prerequisites

First of all, we need to reach some prerequisites in order to install Jogamp.

1. It is necessary to install *Gluegen* and *JOGL* , which are required as compile time dependencies.
2. Our computer must have the following platforms and components

GPU NVidia Geforce >= 8

NVidia GPU Computing SDK

GPU AMD or CPU x86 x86_64 SSE3

Java update version

AMD Accelerated Parallel Processing SDK

CPU Intel

Intel OpenCL SDK

3.2.2 Working on terminal

The following section is explained to work with a Linux system environment.

Once we have all this prerequisites , we are ready to install our Jogamp.

We can Install it following two different ways. In one hand we can build Jogamp plain the terminal or we can install Jogamp in an IDE.

To build Jogamp in our terminal we need to follow the next **steps**:

First we have to **obtain the source code**. We can obtain it using git repositories.

/home/dude/projects/jogamp> git clone git://jogamp.org/srv/scm/jocl.git jocl

Once we obtained the code, we should have three different directories such as gluegen, jogl and jocl inside our root directory jogamp.

The second step will be **build the source code**.
Open a terminal, go inside the jocl directory and type

“ant jar”

The third step will be test our build. Open a terminal and go to the jocl directory and type

“ant test”

The last step will be **build Javadoc**. Open a terminal and go to the jocl directory and type

“ant javadoc”

3.2.3 Working on IDE

In this section , how to install Jogamp in the Eclipse IDE is presented.

It is necessary to create a user library in order to work with Jogamp in Eclipse. It is means that Jogamp library would be link to the project.

The steps to follow are:

1. Go to Window->Preferences

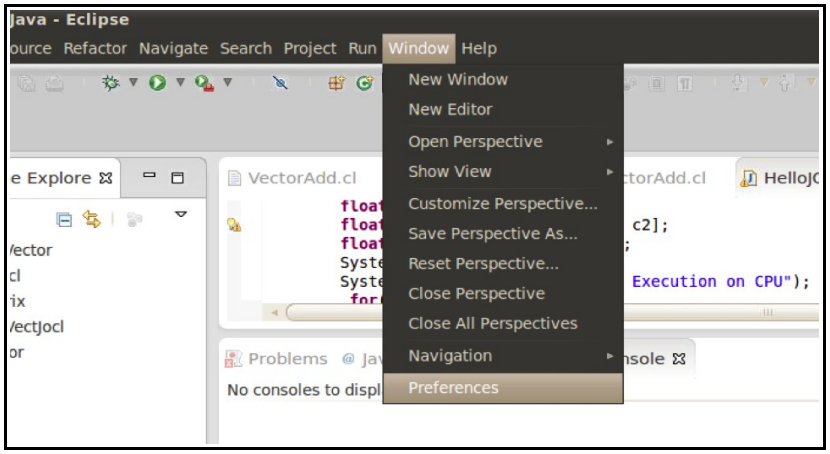

Figure 4: Set up Eclipse with Jogamp I

2. Preferences->Java->Build Path->User Libraries

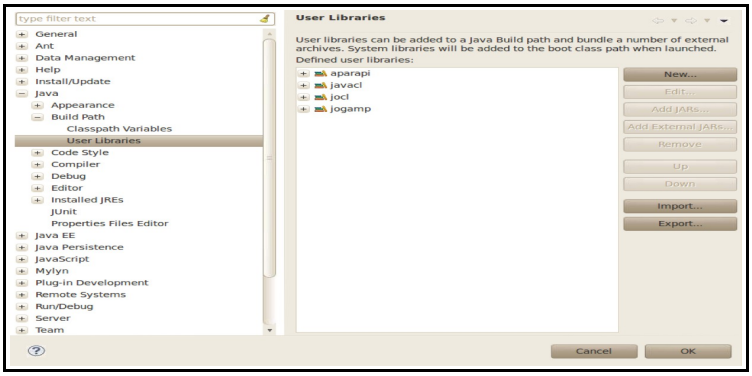

Figure 5: Set up Eclipse with Jogamp II

3. Add new library

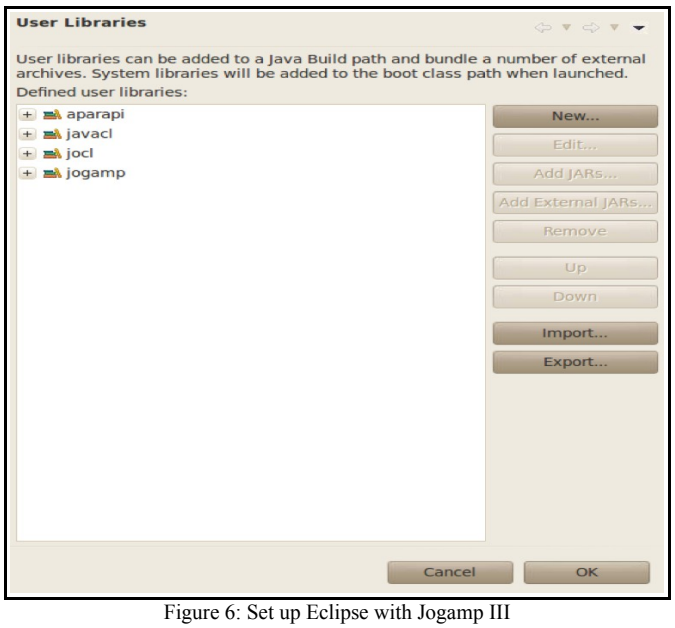

Figure 6: Set up Eclipse with Jogamp III

4. Inside the new library, add external Jars, and look in your directory for the JarFiles of the Jogamp library. The files needed it here are:

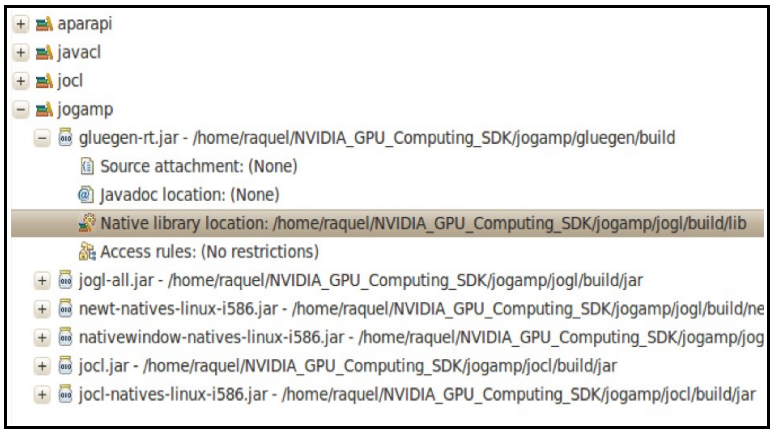

Figure 7: Set up Eclipse with Jogamp IV

-jogl-all.jar , gluegen-rt.jar, newt-natives-linux-i586.jar, nativewindoew-natives-linux-i 568.jar, jocl.jar and jocl-natives-linux-i568.jar

5. Inside every jar file, it is necessary to open the native library location and add the directory of our .so files . In Jogamp, the .so files are saved in the lib directory.

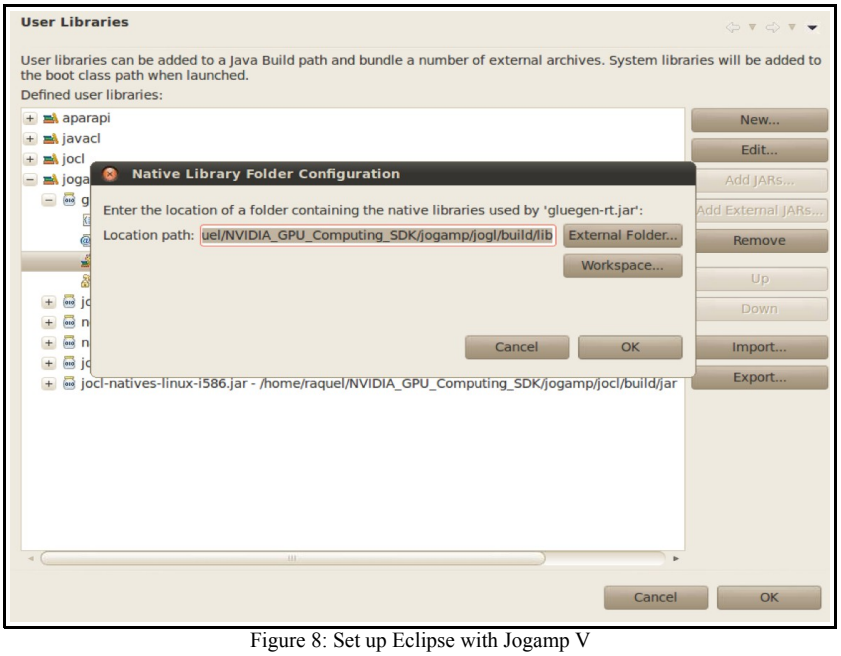

Figure 8: Set up Eclipse with Jogamp V

6. Once the user library is created, it is necessary to link the project with the new library.
    1. Linking in our project , click on Properties

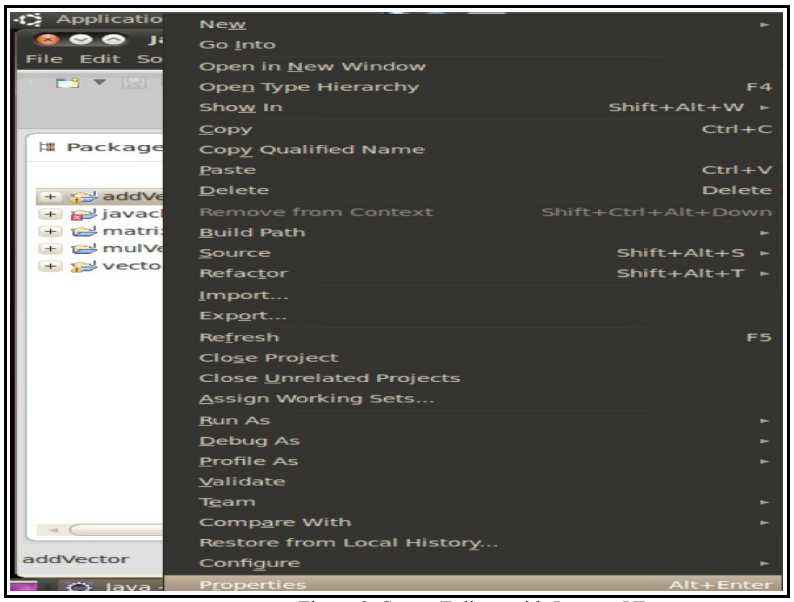

Figure 9: Set up Eclipse with Jogamp VI

2. Click in Java Build path and then libraries

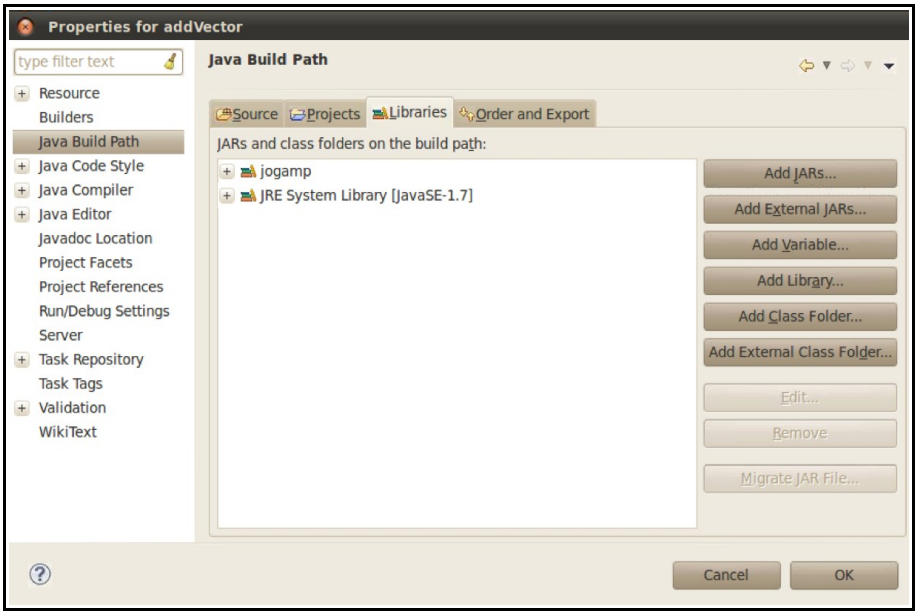

Figure 10: Set up Eclipse with Jogamp VII

3. Click add library, and choose User Library

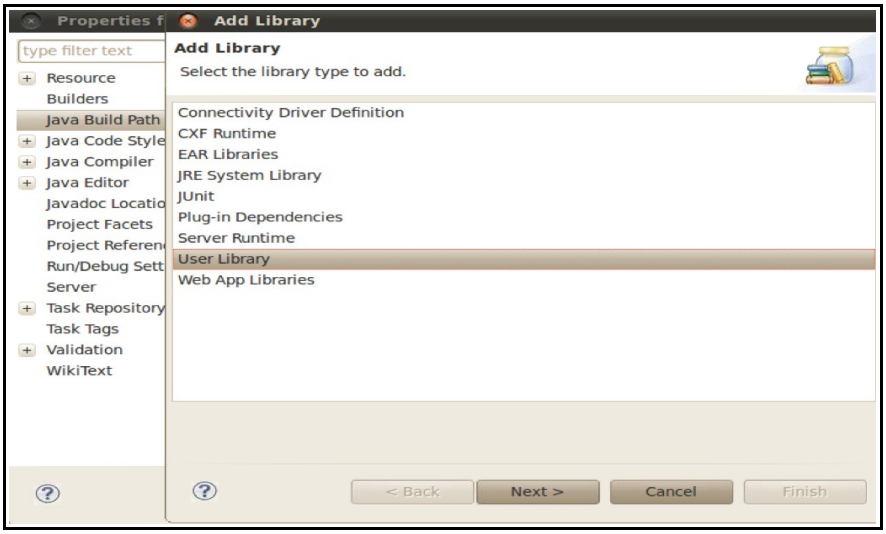

Figure 11: Set up Eclipse with Jogamp VIII

4. Click on the library we want to add, Jogamp library and this case, and click Finish.

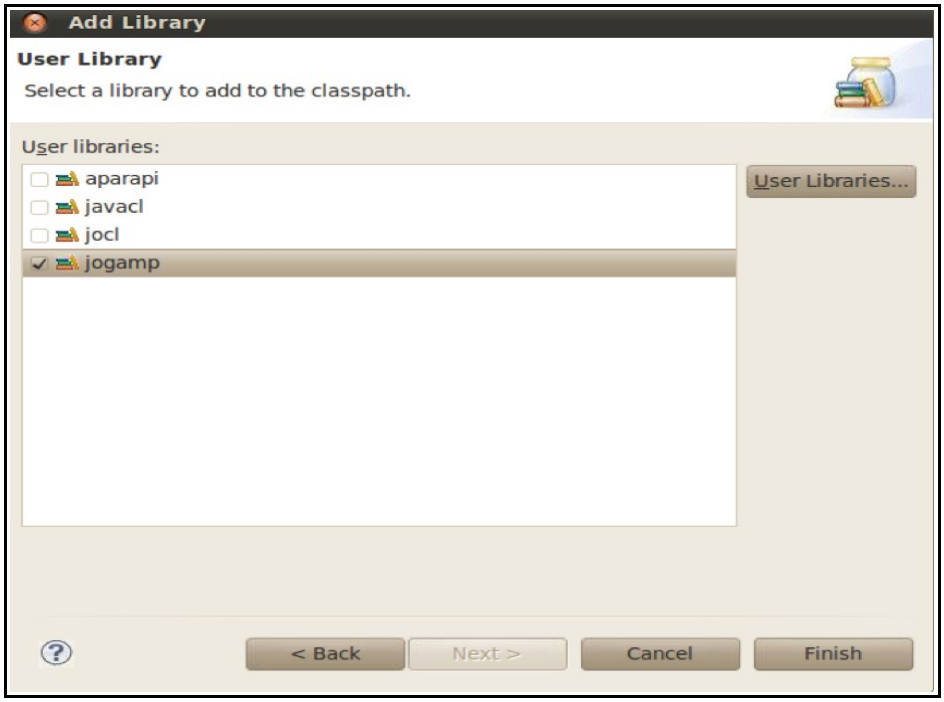

Figure 12: Set up Eclipse with Jogamp IX

5. Finally we can see the Jogamp library linked in our project. IDE eclipse is prepared to build any Jogamp project.

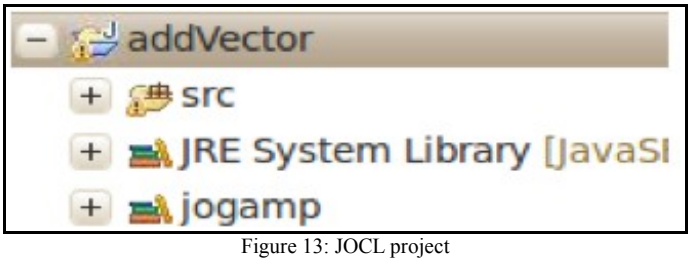

Figure 13: JOCL project

4.JOCL[8]

4.1 Description

This library offers Java-Bindings for OpenCL that are very similar to the original OpenCL API. The functions are provided as static methods, and semantics and signatures of these methods have been kept accordant with the original library functions, exclude for the language-specific limitations of Java.

The OpenCL API may be very tedious at some points, and this is not hidden or simplified, but simply offered by JOCL as it is.

4.2 How to install it

4.2.1 Prerequisites

It does not require install any platform as Jogamp did , just the basic prerequisites such as

GPU NVidia Geforce >= 8
NVidia GPU Computing SDK

GPU AMD or CPU x86 x86_64 SSE3

Java update version

AMD Accelerated Parallel Processing SDK

CPU Intel

Intel OpenCL SDK

4.2.2 Working on terminal

You can download from the JOCL website an example and compile it in the terminal as it is shows in the next example.

```
javac  OpenCLPart1.java  -classpath .;./JOCL-0.1.3a-beta.jar
java  -classpath .;./JOCL-0.1.3a-beta.jar OpenCLPart1
```

```
Obtaining platform...
Test PASSED
Result: [0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

In my experience has been impossible to work on the terminal with JOCL because of multiples errors. It has been much easy working with eclipse IDE.

The 'Trouble Shooting' section on JOCL website mentions one the most frequent error messages when using JOCL .The famous UnsatisfiedLinkError.

It is very important to use the correct DLL for your system. Even if the CPU supports 64 but, it may not be running Java in 64 bit mode. For my any way, it has been unsatisfied.

4.2.3 Working on IDE

In order to work on any IDE such an Eclipse with JOCL, it is necessary to follow the sames steps of the section 3.2.

In this case, only a jar file is needed. The jar file is named as JOCL-01.7.jar and it must include the location of the .so files in the native library location as it can be seen in the next figure.

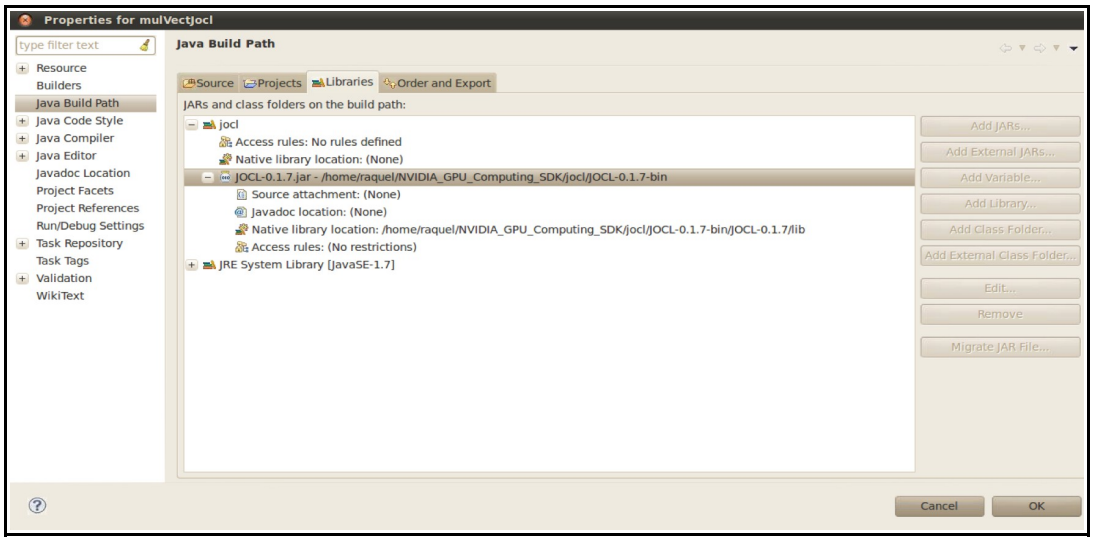

Figure 14: Set up Eclipse with JOCL

5. JavaCL[9]

5.1 Description

This library offers an object-oriented abstraction of OpenCL for Java. It has a low-level interface which is based on JNA and generated using the JNAerator library. The low-level interface serves as the basis for the object-oriented wrapper but is not intended to be used by clients. JavaCL is part of the NativeLibs4Java project, which also contains ScalaCL, a library for accessing OpenCL with Scala.

NativeLibs4Javais a binding JNA developed and maintains a complicated interface due to the following problems:

- The JNA automatic generator creates several options for each feature covered. It is difficult for the developer to decide which one is correct in each case.
- It is necessary that the user knows JNA to use because it uses own classes for use JNA.
- There are few examples available.

5.2 How to install it

5.2.1 Prerequisites

It does not require install any platform as Jogamp did , just the basic prerequisites such as

GPU NVidia Geforce >= 8
NVidia GPU Computing SDK

GPU AMD or CPU x86 x86_64 SSE3

Java update version

AMD Accelerated Parallel Processing SDK

CPU Intel

Intel OpenCL SDK

5.2.2 Working on terminal

The steps to build any JavaCL are showed here:

1. First, install Maven.

2. Checkout nativelibs4java files :

    git clone git://github.com/ochafik/nativelibs4java.git

    cd nativelibs4java/libraries

3. Run the following Maven command :

    mvn install -DskipTests

4. After the build is finished, you'll find JavaCL's full self-contained JAR in OpenCL/JavaCL/target/javacl-xxx-shaded.jar

5. Incremental builds (after a first full build)

6. To build JavaCL Demos, just cd to the libraries/OpenCL/Demos directory and run the following command (works the same for any other sub-project) :

    mvn clean install

To avoid launching tests, you can append a -Dmaven.test.skip=true argument to that command.

5.2.3 Working on IDE

In order to work on any IDE such an Eclipse with JOCL, it is necessary to follow the sames steps of the section 3.2.

In this case, only a jar file is needed. The jar file is named as JOCL-01.7.jar .It is necessary to add i the location of the .so files in the native library location as it shows in the figure below.

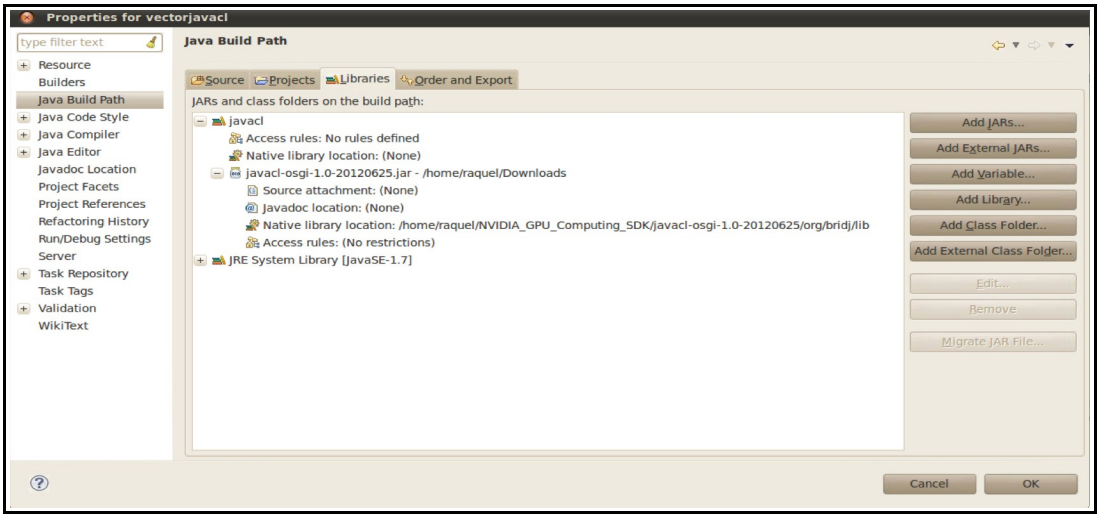

Figure 15: Set up Eclipse with JavaCL

6. Library characteristics

In this section how the three different java binding have been tested is showed.
Vector multiplication has been the baseline application for test every java binding.
This section will show productivity/easy to use versus performance of every java binding.

This section will show all the problems Obtained when trying to install the three different platforms. It will show which platform has been more friendly and easier to learn.
Below, The different characteristics of each platform are discussed here.

6.1 Environment

All the test are realized in the following environment :

-GPU NVidia GeforceGTX 480

-Ubuntu 10.04.03

-Eclipse Juno

6.2 Jogamp class

The Jogamp project is composed by two classes. One class is a Java class called MulJogamp.java which contains all the Java code necessary to implement the vector multiplication and the second class it is the open kernel function multiplication . In order to work with Jogamp library , it is necessary to install also jocl, and gluegen library in our system as we can see in the next figure.

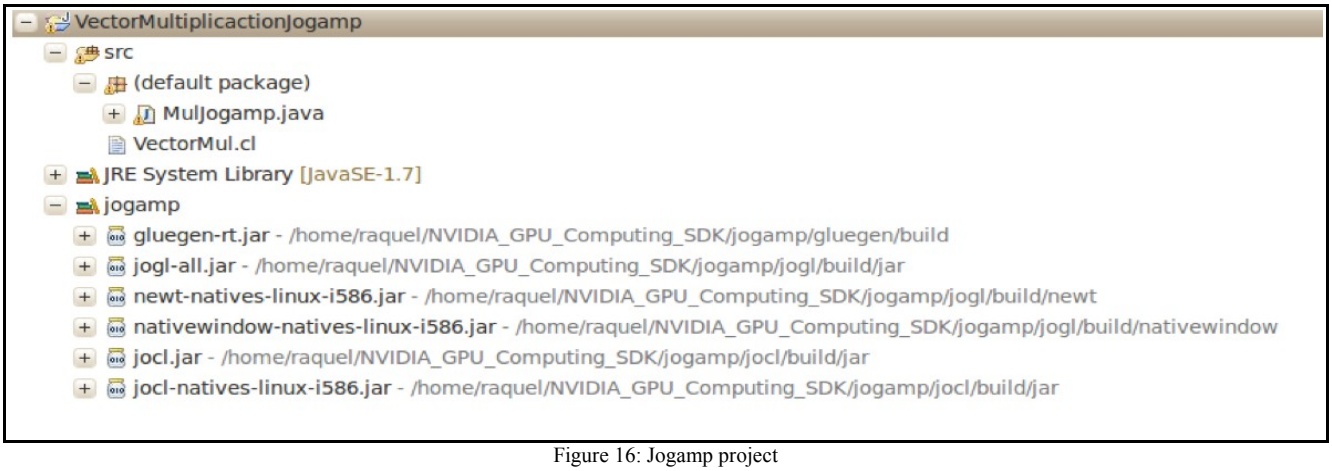

Figure 16: Jogamp project

The figures bellow showed both classes.

VectorMul.cl

```
1
2   // OpenCL Kernel Function for element by element vector multiplication
3   kernel void VectorMul(global const float* a, global const float* b, global float* c, int numElements) {
4
5       // get index into global data array
6       int iGID = get_global_id(0);
7
8       // bound check (equivalent to the limit on a 'for' loop for standard/serial C code
9       if (iGID >= numElements)  {
10          return;
11      }
12
13      // add the vector elements
14      c[iGID] = a[iGID] * b[iGID];
15  }
```

Figure 17: VectorMul.cl class

MulJogamp.java

The java class in composed by two algorithm. One algorithm is using the Jogamp library and the other one is using pure Java. The first algorithm will measure the GPU results and the second one the GPU results.

The algorithm is a vector multiplication of random number. The length of both vectors are 1444777.

```
import com.jogamp.opencl.CLBuffer;
import com.jogamp.opencl.CLCommandQueue;
import com.jogamp.opencl.CLContext;
import com.jogamp.opencl.CLDevice;
import com.jogamp.opencl.CLKernel;
import com.jogamp.opencl.CLProgram;
import java.io.IOException;
import java.nio.FloatBuffer;
import java.util.Random;

import static java.lang.System.*;
import static com.jogamp.opencl.CLMemory.Mem.*;
import static java.lang.Math.*;

public class MulJogamp {

    public MulJogamp(int r, int c1, int c2) {
        // TODO Auto-generated constructor stub
    }

    public static void main(String[] args) throws IOException {

        // set up (uses default CLPlatform and creates context for all devices)
        CLContext context = CLContext.create();
        out.println("created " + context);
```

Figure 18: JOGAMP class I

```
ways make sure to release the context under all circumstances


/ select fastest device
LDevice device = context.getMaxFlopsDevice();
ut.println("using " + device);
/ create command queue on device.
LCommandQueue queue = device.createCommandQueue();
nt elementCount = 1444777; // Length of arrays to process
nt localWorkSize = min(device.getMaxWorkGroupSize(), 256);
nt globalWorkSize = roundUp(localWorkSize, elementCount);
ut.println("local " + localWorkSize);
ut.println("global " + globalWorkSize);
/ load sources, create and build program
LProgram program = context.createProgram(
       MulJogamp.class.getResourceAsStream("VectorMul.cl"))
       .build();
/ A, B are input buffers, C is for the result
LBuffer<FloatBuffer> clBufferA = context.createFloatBuffer(
       globalWorkSize, READ_ONLY);
LBuffer<FloatBuffer> clBufferB = context.createFloatBuffer(
       globalWorkSize, READ_ONLY);
LBuffer<FloatBuffer> clBufferC = context.createFloatBuffer(
       globalWorkSize, WRITE_ONLY);

```

Figure 19: JOGAMP class II

```
            out.println("used device memory: "
                    + (clBufferA.getCLSize() + clBufferB.getCLSize() + clBufferC
                            .getCLSize()) / 1000000 + "MB");

            // fill input buffers with random numbers
            // (just to have test data; seed is fixed -> results will not change
            // between runs).
            fillBuffer(clBufferA.getBuffer(), 12345);
            fillBuffer(clBufferB.getBuffer(), 67890);

            // get a reference to the kernel function with the name 'VectorAdd'
            // and map the buffers to its input parameters.
            CLKernel kernel = program.createCLKernel("VectorMul");
            kernel.putArgs(clBufferA, clBufferB, clBufferC)
                    .putArg(elementCount);

            // asynchronous write of data to GPU device,
            // followed by blocking read to get the computed results back.
            long time = nanoTime();
            queue.putWriteBuffer(clBufferA, false)
                    .putWriteBuffer(clBufferB, false)
                    .put1DRangeKernel(kernel, 0, globalWorkSize, localWorkSize)
                    .putReadBuffer(clBufferC, true);
            time = nanoTime() - time;

```

Figure 20: JOGAMP class III

```
76              // print first few elements of the resulting buffer to the console.
77              out.println("a*b=c results snapshot: ");
78              for (int i = 0; i < 10; i++)
79                  out.print(clBufferC.getBuffer().get() + ", ");
80              out.println("...; " + clBufferC.getBuffer().remaining() + " more");
81
82              out.println("computation in GPU took: " + (time / 1000000) + "ms");
83
84              final int r = 1444777;
85              final int c1 = 1444777;
86              final int c2 = 1444777;
87
88              MulJogamp ap = new MulJogamp(r, c1, c2);
89              long time1 = System.currentTimeMillis();
90              ap.normalMatMulCalc();
91              out.println("Time taken for kenel execution in Sequential CPU mode is :"
92                      + (System.currentTimeMillis() - time1));
93
94          } finally {
95              // cleanup all resources associated with this context.
96              context.release();
97          }
98
99      }
100
```

Figure 21: JOGAMP class IV

```
101     private static void fillBuffer(FloatBuffer buffer, int seed) {
102         Random rnd = new Random(seed);
103         while (buffer.remaining() != 0)
104             buffer.put(rnd.nextFloat());
105         buffer.rewind();
106     }
107
108     private static int roundUp(int groupSize, int globalSize) {
109         int r = globalSize % groupSize;
110         if (r == 0) {
111             return globalSize;
112         } else {
113             return globalSize + groupSize - r;
114         }
115     }
116
117     public void normalMatMulCalc() {
118
119         final int r = 1444777;
120         final int c1 = 1444777;
121         final int c2 = 1444777;
122
123         float[] matA = new float[c1];
124         float[] matB = new float[c2];
125         float[] matC = new float[c1 * c2];
126         float[] C = new float[c1 * c2];
```

Figure 22 : JOGAMP class V

```
127
128         out.println("Sequential Execution on CPU");
129         for (int i = 0; i < r; i++) {
130
131             matA[i] = new Random().nextFloat();
132             matB[i] = new Random().nextFloat();
133             C[i] = matA[i] * matB[i];
134
135         }
136         for (int i = 0; i < 10; i++) {
137             System.out.print(C[i]);
138
139         }
140
141     }
142 }
143
```

Figure 23: JOGAMP class VI

6.2.1 Jogamp characteristics

1.Development time

-Get used to how to use the library	15 h
-Implementation of the class	5 h
-Test	2 h
-Total	22 h

2. Run time of an algorithm

In the test realize the results are :7milliseconds and 17MB of used memory for the algorithm with 14447777 length of vectores.

3. Complexity

The Jogamp class is composed by 145 +15 lines of code. Moreover , around 30 lines are from the pure java algorithm.

At the end , I used 160 lines in order to create my baseline class in Jogamp.

4. Platforms support

-It is based on Java so it is supported for any platform with supports Java and Opencl.
-JOGAMP needs Gluegen and JOGL to work.

5. Documentation

The documentation found on the wiki is not complete. It has been difficult to start to program with JOGAMP because of the difficult of work on the terminal. Several errors showed up and it is not easy to fix then on the terminal. It has been much easier to work on an IDE but the documentation does not show how to work on any IDE such as eclipse or netbeans. The most wasting time has been how to set up our IDE to work with JOGAMP. But after this project, how to set up your IDE eclipse is showed.

6. Status

I did not success installing the platform on the terminal. Several errors showed up and I could not fix them.

6.3 JOCL class

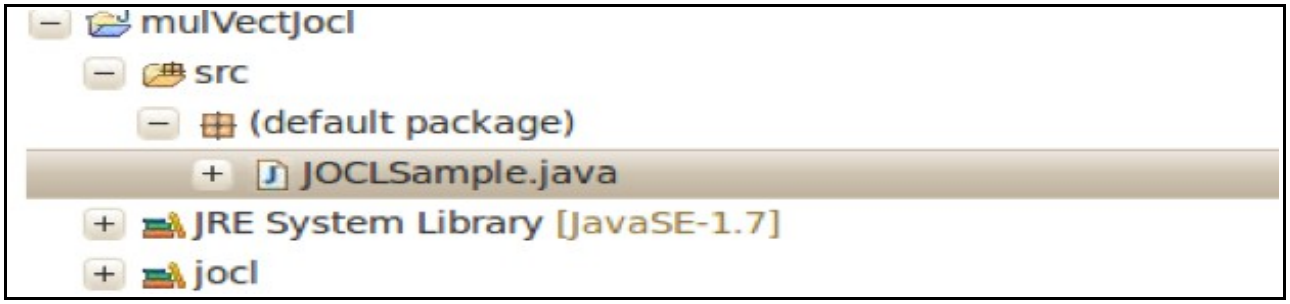

Figure 24: Jocl project

```
1  import static java.lang.System.nanoTime;
2  import static java.lang.System.out;
3  import static org.jocl.CL.*;
4  import java.util.Random;
5  import org.jocl.*;
6  public class JOCLSample
7  {     /**
8       * The source code of the OpenCL program to execute
9       */
10     private static String programSource =
11         "__kernel void "+
12         "sampleKernel(__global const float *a,"+
13         "             __global const float *b,"+
14         "             __global float *c)"+
15         "{"+
16         "    int gid = get_global_id(0);"+
17         "    c[gid] = a[gid] * b[gid];"+
18         "}";
19     public static void main(String args[])
20     { // Create input- and output data
21         int n = 1444777;
22         float srcArrayA[] = new float[n];
23         float srcArrayB[] = new float[n];
24         float dstArray[] = new float[n];
25         for (int i=0; i<n; i++)
```

Figure 25: JOCL class I

```
26         {
27
28             srcArrayA[i]= new Random().nextFloat();
29             srcArrayB[i] = new Random().nextFloat();
30             dstArray[i] = srcArrayA[i]*srcArrayB[i];
31
32         }
33         Pointer srcA = Pointer.to(srcArrayA);
34         Pointer srcB = Pointer.to(srcArrayB);
35         Pointer dst = Pointer.to(dstArray);
36
37         // The platform, device type and device number
38         // that will be used
39         final int platformIndex = 0;
40         final long deviceType = CL_DEVICE_TYPE_ALL;
41         final int deviceIndex = 0;
42
43         // Enable exceptions and subsequently omit error checks in this sample
44         CL.setExceptionsEnabled(true);
45
46         // Obtain the number of platforms
47         int numPlatformsArray[] = new int[1];
48         clGetPlatformIDs(0, null, numPlatformsArray);
49         int numPlatforms = numPlatformsArray[0];
50
```

Figure 26: JOCL class II

```
51          // Obtain a platform ID
52          cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
53          clGetPlatformIDs(platforms.length, platforms, null);
54          cl_platform_id platform = platforms[platformIndex];
55
56          // Initialize the context properties
57          cl_context_properties contextProperties = new cl_context_properties();
58          contextProperties.addProperty(CL_CONTEXT_PLATFORM, platform);
59
60          // Obtain the number of devices for the platform
61          int numDevicesArray[] = new int[1];
62          clGetDeviceIDs(platform, deviceType, 0, null, numDevicesArray);
63          int numDevices = numDevicesArray[0];
64
65          // Obtain a device ID
66          cl_device_id devices[] = new cl_device_id[numDevices];
67          clGetDeviceIDs(platform, deviceType, numDevices, devices, null);
68          cl_device_id device = devices[deviceIndex];
69
70          // Create a context for the selected device
71          cl_context context = clCreateContext(
72              contextProperties, 1, new cl_device_id[]{device},
73              null, null, null);
74
75          // Create a command-queue for the selected device
```

Figure 27: JOCL class III

```
76          cl_command_queue commandQueue =
77              clCreateCommandQueue(context, device, 0, null);
78
79          // Allocate the memory objects for the input- and output data
80          cl_mem memObjects[] = new cl_mem[3];
81          memObjects[0] = clCreateBuffer(context,
82              CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
83              Sizeof.cl_float * n, srcA, null);
84          memObjects[1] = clCreateBuffer(context,
85              CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
86              Sizeof.cl_float * n, srcB, null);
87          memObjects[2] = clCreateBuffer(context,
88              CL_MEM_READ_WRITE,
89              Sizeof.cl_float * n, null, null);
90
91          out.println("used device memory: "
92                  + (n*Sizeof.cl_float)/1000000+ "MB");
93
94          // Create the program from the source code
95          cl_program program = clCreateProgramWithSource(context,
96              1, new String[]{ programSource }, null, null);
97
98          // Build the program
99          clBuildProgram(program, 0, null, null, null, null);
100
```

Figure 28: JOCL class IV

```
101         // Create the kernel
102         cl_kernel kernel = clCreateKernel(program, "sampleKernel", null);
103
104         long time = nanoTime();
105         // Set the arguments for the kernel
106         clSetKernelArg(kernel, 0,
107             Sizeof.cl_mem, Pointer.to(memObjects[0]));
108         clSetKernelArg(kernel, 1,
109             Sizeof.cl_mem, Pointer.to(memObjects[1]));
110         clSetKernelArg(kernel, 2,
111             Sizeof.cl_mem, Pointer.to(memObjects[2]));
112
113         // Set the work-item dimensions
114         long global_work_size[] = new long[]{n};
115         long local_work_size[] = new long[]{1};
116
117         // Execute the kernel
118         clEnqueueNDRangeKernel(commandQueue, kernel, 1, null,
119             global_work_size, local_work_size, 0, null, null);
120
121         // Read the output data
122         clEnqueueReadBuffer(commandQueue, memObjects[2], CL_TRUE, 0,
123             n * Sizeof.cl_float, dst, 0, null, null);
124
125         time = nanoTime() - time;
```

Figure 29: JOCL class V

```
126         out.println("computation in GPU took: "+(time/1000000)+"ms");
127
128         // Release kernel, program, and memory objects
129         clReleaseMemObject(memObjects[0]);
130         clReleaseMemObject(memObjects[1]);
131         clReleaseMemObject(memObjects[2]);
132         clReleaseKernel(kernel);
133         clReleaseProgram(program);
134         clReleaseCommandQueue(commandQueue);
135         clReleaseContext(context);
136
137         for (int i=0; i<10; i++){
138
139             System.out.print(dstArray[i]);
140
141         }
142     }
143 }
144
```

Figure 30: JOCL class VI

6.3.1 Jogamp characteristics

1.Development time [devel].

-Get used to how to use the library	10 h
-Implementation of the class	5 h
-Test	2 h
-Total	17 h

2. Run time of an algorithm [time].

In the test realize the results are : 16milliseconds and 5 MB of used memory for the algorithm with 14447777 length of vectors.

3. Complexity, ability to utilize all device features [complex].

The Jocl class is composed by 144 lines of code.

4. Platforms support

-It is based on Java so it is supported for any platform with supports Java and Opencl.

5. Documentation

The documentation found on the wiki is not complete. It has been difficult to start to program with JOCL because of the difficult of work on the terminal. Several errors showed up and it is not easy to fix then on the terminal. It has been much easier to work on an IDE as JOGAMP case. It has been less wasted of time because JOCL follows the sames steps as JOGAMP to set up the IDE platform.

6. Status

The 'Trouble Shooting' section on JOCL website mentions one the most frequent error messages when using JOCL .The famous UnsatisfiedLinkError.

It is very important to use the correct DLL for your system. Even if the CPU supports 64 but, it may not be running Java in 64 bit mode. For my any way, it has been unsatisfied.

6.4 JAVACL

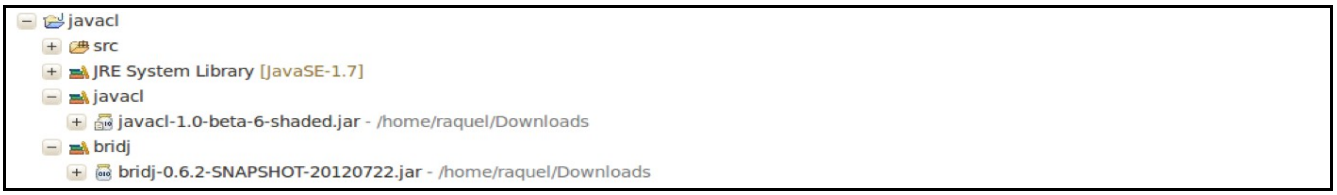

Figure 31: JAVACL project

```
__kernel void add_floats(__global const float* a, __global const float* b, __global float* out, int n)
{
    int i = get_global_id(0);
    if (i >= n)
        return;

    out[i] = a[i] * b[i];
}
```
Figure 32: Vectormultiplicacion.cl javacl

```
import java.io.IOException;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.util.Random;
import com.nativelibs4java.opencl.*;
import com.nativelibs4java.opencl.CLMem.Usage;
import com.nativelibs4java.util.*;
import org.bridj.Pointer;
import static org.bridj.Pointer.*;
import static java.lang.System.nanoTime;

public class JAVACLMultiplication {
    public static void main(String[] args) throws IOException, CLBuildException {
        CLContext context = JavaCL.createBestContext();
        CLQueue queue = context.createDefaultQueue();
        ByteOrder byteOrder = context.getByteOrder();

        int n = 1444777;
        Pointer<Float>
            aPtr = allocateFloats(n).order(byteOrder),
            bPtr = allocateFloats(n).order(byteOrder);
     for (int i = 0; i < n; i++) {

            aPtr.set(i,new Random().nextFloat());
            bPtr.set(i,new Random().nextFloat());
        }
```
Figure 33: JAVACLMultiplication.java class

```
        }

        CLBuffer<FloatBuffer> a = context.createFloatBuffer(Usage.Input,n);
        CLBuffer<FloatBuffer> b = context.createFloatBuffer(Usage.Input,n);

        CLBuffer<FloatBuffer> out = context.createFloatBuffer(Usage.Output,n);

        System.out.println("used device memory: "
                + (a.getByteCount()+b.getByteCount()+out.getByteCount())/1000000 +"MB");

        // Read the program sources and compile them :
        String src = IOUtils.readText(JAVACLMultiplication.class.getResource("VectorMultiplication.cl"));
        CLProgram program = context.createProgram(src);
        // Get and call the kernel :
        long time = nanoTime();
        CLKernel addFloatsKernel = program.createKernel("add_floats");
        addFloatsKernel.setArgs(a, b, out, n);
        CLEvent addEvt = addFloatsKernel.enqueueNDRange(queue, new int[] { n });
        FloatBuffer outPtr = out.read(queue, addEvt); // blocks until add_floats finished
        time = nanoTime() - time;
        System.out.println("computation in GPU took: "+(time/1000000)+"ms");
        // Print the first 10 output values :
        for (int i = 0; i < 10 && i < n; i++)
            System.out.println("out[" + i + "] = " + outPtr.get(i));
       }
}
```
Figure 34: JAVACLMultiplication.java II class

6.4.1 Javacl characteristics

1.Development time

-Get used to how to use the library 5 h
-Implementation of the class 4 h
-Test 2 h

-Total 11 h

2. Run time of an algorithm

In the test realize the results are : 20 milliseconds and 17 MB of used memory for the algorithm with 14447777 length of vectors.

3. Complexity, ability to utilize all device features

The Javacl class is composed by 51 lines of code.

4. Platforms support

-It is based on Java so it is supported for any platform with supports Java and Opencl.
- To work with Javacl has been necessary to instal a new library bridj to use Pointer as we can see in the figure 31.

5. Documentation

The documentation found on the wiki is was complete. It has been easy to start to program with Javacl. All the information about how to install it on the terminal or on the IDE was on the wiki. Moreover, I found several examples and tutorial on the wiki , which have been very helpful.

6. Status

	JOGAMP	JOCL	JAVACL
Developed time	17	22	11
Run time	16 ms 5 MB	7 ms 17 MB	20 ms 17 MB
Complexity	144	160	51
Platforms	No dependecies	Dependence of gluegen and jogl	Dependence of bridj
Documentation	Not enough	Not enough	complete
Bugs	Yes	Yes	no

List 7: Characteristics of jocl, jogamp and javacl.

7.Experiments

This section shows the results obtained from diferents tests. We begin with a test that measured the execution time on a CPU and a GPU.

7.1 Cpu and Gpu Test

The first test shows the comparison result of the same algorithm working under the same library in CPU and GPU. The conditions of the test are:

-The algorithm is a vector multiplication, where the vector is filled in with random numbers.

-The Jogamp java class is composed by two algorithms. The first one is a vector multiplication working with Jogamp library and the second one is an algorithm working with pure java. The first algorithm will show the result of work with GPU and the second one with CPU.

7.1.1 JOGAMP time CPU vs GPU

The graphic below shows the time measured in CPU and GPU working under the same library (JOGAMP) and algorithm (vector multiplication) with severals lengths of vector.

length of vector	cpu	gpu
0	0	0
100	1	0
1000	5	0
10000	17	0
100000	69	0
1000000	538	5
10000000	4970	46

List 2: Time jogamp cpu vs gpu

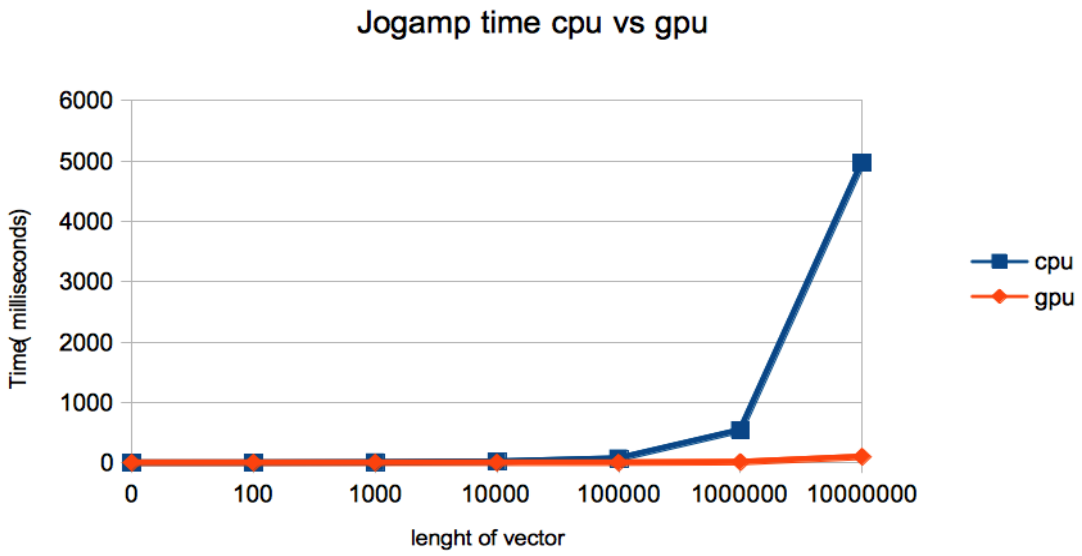

Figure 35: Jogamp time CPU vs GPU

The figure 35 shows that the time is stable for CPU and GPU until a length of 100000.

Once the length of the vector becomes over 100000, the execution time on a CPU increases significantly. At the end, the time used for a CPU is 50 times over the time used by a GPU.

7.1.2 JOCL time CPU vs GPU

The second graphic below shows the time measured in CPU and GPU working under the same library (JOCL) and algorithm (vector multiplication) with severals lengths of vector.

length of vector	cpu	gpu
0	0	0
100	1	0
1000	5	0
10000	17	0
100000	69	1
1000000	538	10
10000000	4970	97

List 3: Time jocl cpu vs gpu

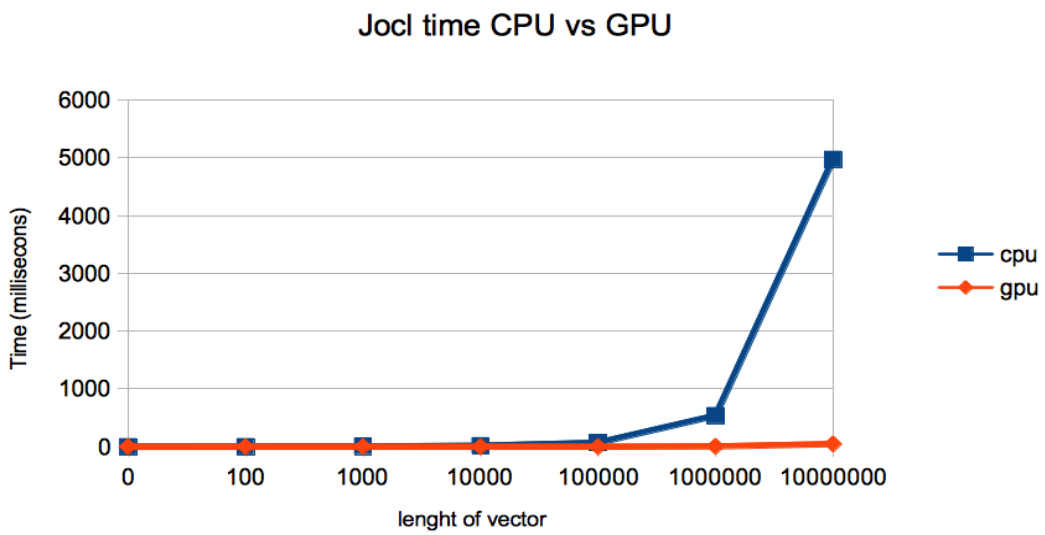

Figure 36: JOCL time CPU vs GPU

The first graphic shows that the time is stable for CPU and GPU until a length of 100000.

Once the length of the vector is over 100000, the time for CPU increase in a extremely mode as the same situation of JOGAMP.

At the end, the time used for CPU is 100 times over the time used by GPU.

7.1.3 JAVACL time CPU vs GPU

The third graphic below shows the time measured in CPU and GPU working under the same library (JAVACL) and algorithm (vector multiplication) with severals lengths of vector.

length of vector	cpu	gpu
0	0	0
100	1	10
1000	5	11
10000	17	11
100000	69	11
1000000	538	14
10000000	4970	42

List 4: Time javacl cpu vs gpu

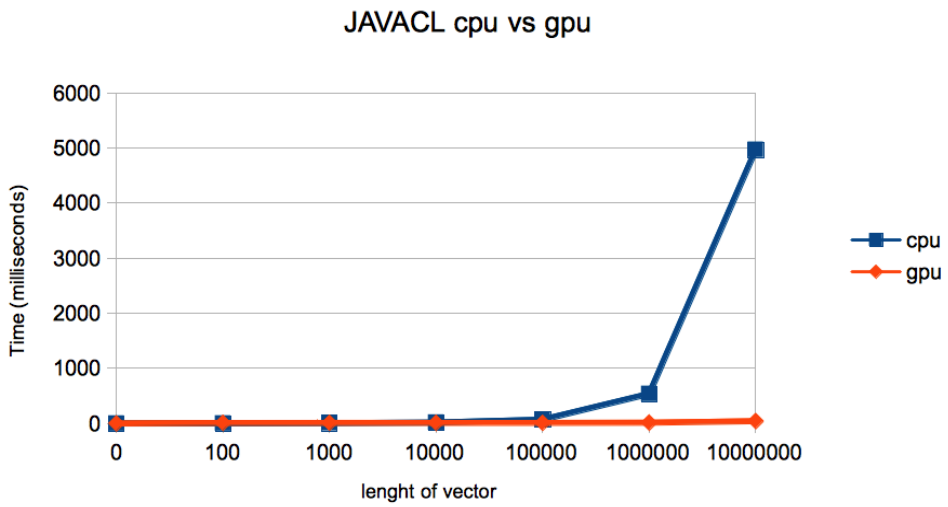

Figure 37: Jogamp time CPU vs GPU

In on hand ,the first test shows that Jocl is less effective in order of time than Jogamp and Javacl. It takes almost the double time to run the same algorithm in the sames circumstances. More test to conformance this affirmation are showed later.

In other hand, Jogamp and Javacl obtained similar result although Jogamp is more effective. For Jogamp it takes less time to run the algorithm, it is faster , as we can see that the time was under 1 ms until a length of 100000.

In order to support this idea, we can take a look at the graphic below.

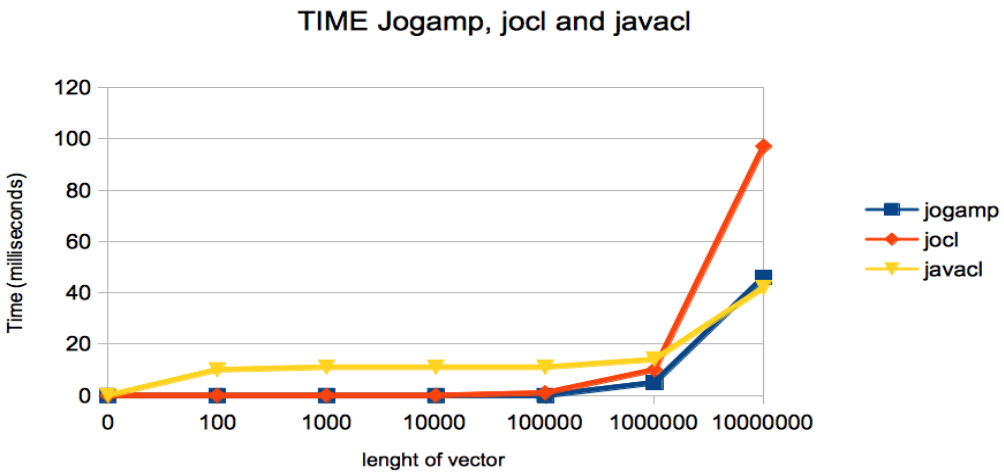

Figure 38: JOCL, JOGAMP AND JAVACL TIME

7.2 Time and Memory test

We measured time and memory on different platforms and the results are shown below.

The graphic below shows the time measured working with JOCL,JOGAMP AND JAVACL using the baseline class vector multiplication. The test shows how the time is changing in order of the length of the vector.

length of vector	JOGAMP	JOCL	JAVACL
0	0	0	0
100	0	0	10
1000	0	0	11
10000	0	0	11
100000	0	1	11
1000000	5	10	14
10000000	46	97	42

List 5: Time jocl, jogamp and javacl.

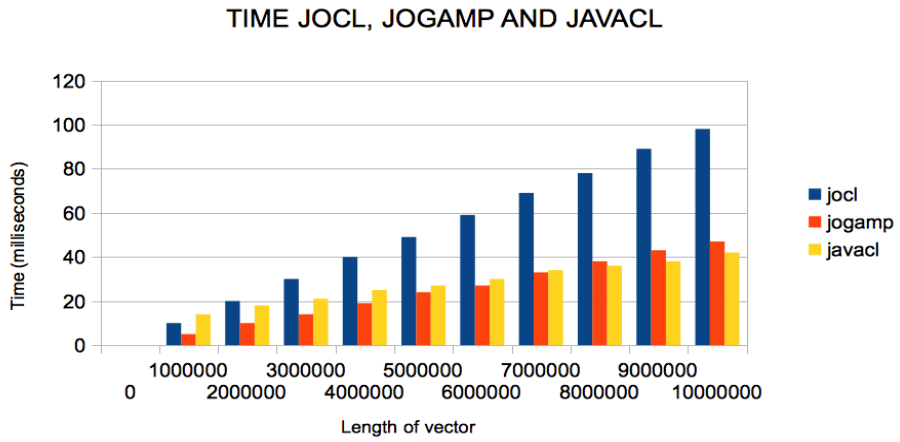

Figure 39: JOCL, JOGAMP AND JAVACL TIME

This graphic shows that Jocl library is less effective in order of time that Jogamp and Javacl.

The graphic shows the difference time obtained on the test. Jocl library needs double time to run the same algorithm under the sames conditions that Jogamp and Javacl library.

The graphic below shows the memory measured working with JOGAMP and JOCL using the baseline class vector multiplication. The test shows how the memory is changing in order of the length of the vector.

Length of array	Memory JOCL (MB)	Memory JOGAMP (MB)	Memory JAVACL (MB)
0	0	0	0
1000000	4	12	12
2000000	8	24	24
3000000	12	36	36
4000000	16	48	48
5000000	20	60	60
6000000	24	72	72
7000000	28	84	64
8000000	32	96	96
9000000	36	108	108
10000000	40	120	120

List 6: Memory jocl, jogamp and javacl.

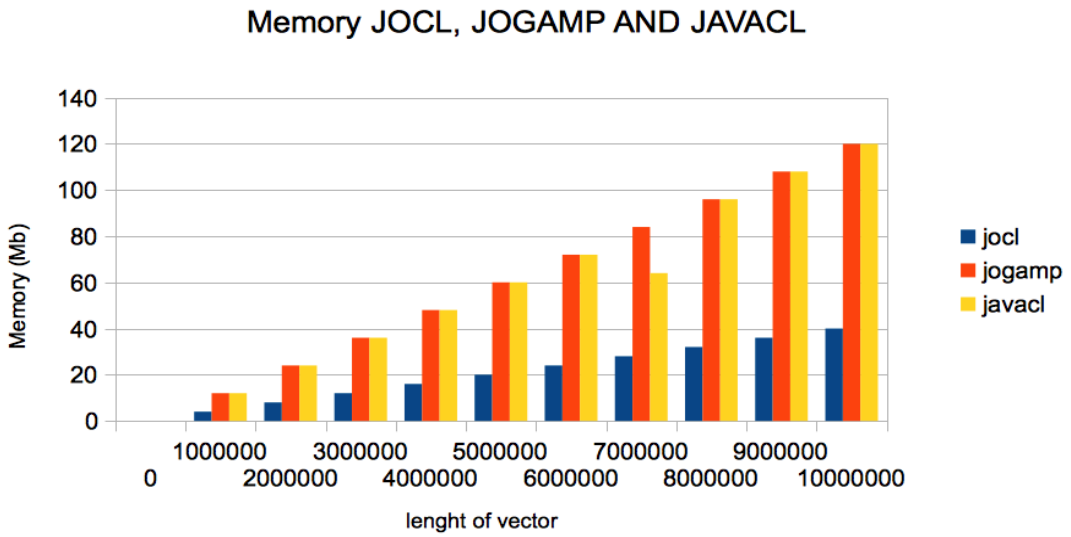

Figure 40: JOCL, JOGAMP AND JAVACL memory

The figure 40 shows that Jogamp and Javacl libraries are less effective in order of Memory that Jocl. The graphics shows the difference memory used obtained on the test. Jogamp and Javac libraries need more than the double memory to run the same algorithm under the sames conditions than Jocl library.

8.Conclusion

For the general conclusion , this section summarize the three different test evaluated during this project.(Section 6 and 7)

First of all, the differences **characteristics of each library**.

As we can see in the resume of List number 7, Jocl is the library that shows better results in order of memory and Jogamp in order of memory, but we have also to admit that both libraries are more complex to work with.

Javacl has been the easiest library to work with. It has three times less code lines and the library and documentation is more friendly than the others. Furthermore, it has been the only succeed library working on the terminal.

Second of all, the **time tests**.

Jogamp showed on the test that is the most effective library in order of time. It was the fastest, twice as the other two libraries tested.

Jocl and Javacl obtained similar results although Jocl is a bit more effective than Javacl but still being twice slow that Jogamp.

Finally, the **memory tests**.

Observing the memory test, Jocl was the most effective library in order of memory. It was the one that used less memory with the same data as the other two libraries.

Moreover , Jogamp and Javacl obtained the same result in order of memory but they still using three times more memory than Jocl.

In conclusion,

In every project to implement working with Java and OpenCL, the programmer has to decide which library fit better with the project depending of the characteristics.

After the tests realized, we can describe Jogamp as the most efficient library if the goal of the project is is be faster. Jocl as the most efficient library saving used memory and Javacl as the most friendly library and as powerful in saving memory as Jocl.

9.Acronyms

GPU	Graphics processing unit
API	Application programming interface
CPU	Central processing unit
FPGA	Fiel programmable gate array
JNI	Java Nativa Interface
GPGPU	General purpose computing on graphics processing units
JVM	Java virtual machine
NIO	New I/O

List 8: Acronyms

10.Bibliography

[1] Opencl. Avaible at <http://www.khronos.org/opencl/>

[2]Tiobe. Avaible at<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.>

[3] Parallel computing . Avaible at <https://computing.llnl.gov/tutorials/parallel_comp/>

[4] Java. Avaible at <http://en.wikipedia.org/wiki/Java_(programming_language)>

[5] GPGPU. Avaible at <http://gpgpu.org>

[6] Cpu vs Gpu. Avaible at <http://blog.goldenhelix.com/?p=374>

[7] Jogamp. Avaible at <http://jogamp.org>

[8] Jocl. Avaible at <http://www.jocl.org>

[9] Javacl. Avaible at <http://code.google.com/p/javacl/>