JAVA GPU: TECHNOLOGIES COMPARISON

Noémien Kocher

A semester project

Department of computer science
University of Applied Sciences of Western Switzerland
Fribourg, Switzerland

Supervisors: Kuonen Pierre, Jean Hennebert, Gonçalves Lourenço Marco José, Beat Wolf

Abstract

The last years have seen the rise of using GPU's not only for graphics computing, but also for a more global use in parallel computing. This completely new field comes with dedicated GPU's (GPGPU, Global Purpose GPU) that are not specialized anymore, but can be used to compute operations normally conducted on a CPU. Along with hardwares comes new programming technologies that enable programmers to use the GPU in their programs. This field is mainly divided between CUDA, which comes from NVIDIA, and OpenCL, which is an open standard. In this paper, we will be looking at technologies that enable programmers to perform GPU operations from Java code so as to evaluate them. The first chapter (*Introduction*) will select the candidates and set the test battery. The second one (*Analysis*) will evaluate the candidates according to our test battery. The third chapter (*Results*) will compare the results obtained between the candidates and the last chapter (*Conclusion*) will finally explain which solution is better and in which cases.

Keywords. Java, GPU, CUDA, OpenCL, Aparapi, JCuda, Benchmark

Contents

List of Figures

Chapter 1

Introduction

This chapter will be focusing on defining the needs, selecting the candidates and setting up a test battery that will be used to compare the solutions.

Based on defined criteria, the test battery should be able to provide a fair comparison of the chosen solutions, giving a similar systematic test approach for each one.

The candidate selection, again based on defined criteria, should be able to rank every potential candidate based on an individual mark.

1.1 Goals

This project aims to compare different solutions that make GPU programming available in Java. Our measures will be based on the execution speed and ease of use.

We basically need either Java frameworks that give GPU computing access via some API's or solutions that directly convert Java bytecode so it is runnable on a GPU.

1.2 Pre-analysis

This chapter will be focused on selecting the candidates we want to use for our comparisons. Based on some criteria, we will run through each candidate (see appendix C) we found and look if they are interesting enough to be chosen. Our criteria will be :

- Time since the last update
- Adoption level, community activity and documentation
- Adequacy for our needs

1.2.1 Time since last update

If available, the time since the last update will be the last commit. If there is no public SVN, repository or possibilities to find the last code commit, we will be looking at the date of the last product release. We will take the most recent date found.

1.2.2 Adoption level, community activity and documentation

For those criteria, mostly subjective, we will be looking at :

- Quantity of pertinent search results in Google
- Commit activity (if available)
- Forum activity (if exists)
- Quantity of comments, articles and blogs found

1.2.3 Adequacy

For this criteria, we will be looking at the functionalities provided and see if it meets the needs of our project. Basically, it should be able to provide GPU computing capabilities via Java programming.

1.2.4 Pre-analysis table

Time from update	**Adoption level, community activity & documentation**	**Adequacy**
aparapi		
3 months	Aparapi, developed by AMD, appears often in articles and researches. Seems to be in a very high adoption level and has a good documentation.	Allow GPU programming via bytecode translation for OpenCL.
rootbeer		
9 months	According to the readme on Github, it is still in a pre-production beta. It doesn't claim to offer a simple GPU programming tool giving an easily high speed up. Was under active development last year but seems to be left with no recent updates. It is having its longest time of inactivities since its beginning. Documentation not very elaborated.	Allow GPU programming by converting bytecode to CUDA programs.
java-gpu		
6 years	Doesn't seem to be maintained anymore. Has no documentation and no recent version.	Allow GPU programming by translating bytecode into CUDA code.
ScalaCL		
A year	Development stopped a year ago. Not actively maintained, seems to be dying.	Offers GPU programming via Scala and not directly with Java.
JavaCL		
6 months	Has a decent documentation but no recent activities this past years. The activities graph on github seems to show that the project is dying.	An OpenCL wrapper for Java.

jocl		
A year	It has its own website and forum, which is not actively used (only 1 post this year so far). Doesn't seem to have a solid documentation.	Java binding 1:1 mapping to OpenCL
lwjgl		
today	It is a popular framework with a solid documentation and an active forum.	This is a game framework that is not primarily suited for GPU programming.
jcuda		
Last month	Has it's own website and forum, which is not very active. Has some tutorials and an api documentation that seems well.	Translate special Java code to CUDA-C.
jacc		
This year	Is not available and still under development.	Java framework that performs on low-level at runtime to use the GPU.
CUDA4J		
Last year	Is updated to Java 8 and developed by IBM, which certainly means a long time support.	Java programming API that uses lambdas and streams.
PJ2		
Last month	It is an API developed at the Rochester Institute. It is still under maintenance but doesn't have a community and is not really well known.	Middleware API for Java programming on multicore platforms.

1.2.5 Grades

To rank our options, we will give, based on each criteria, a grade going from 0 to 10, 10 being the best and 0 the worst.

Time from update, adoption level and adequacy grade will go according to the following tables:

Table 1.2: Grade: Time from update

< 3 months	< 6 months	< 1 years	< 2 years	> 2 years
10	8	5	3	0

Table 1.3: Grade: Adoption level, community activity and documentation

Is well referenced, found many articles and is well documented	Appears quite often in a google search, found some articles and has a fairly good documentation	Found some results in a google search and has a documentation	Didn't found much results in a google search, has a poor documentation
10	8	5	3

Table 1.4: Grade: Adequacy

Fits perfectly our needs	Fits partially our needs, or the solution is not aimed to particularly fits out needs	Doesn't fit our needs
10	3	0

1.2.6 Grades table

According to our pre-analysis (see section 1.2.4) and our grades (section 1.2.5), each candidates has been evaluated and got a corresponding mark. Based on the marks, we will choose the candidates that have the highest ones.

	Time from update	Adoption level, community activity & documenation	Adequacy	**Grade**
aparapi	10	10	10	**30**
jcuda	10	8	10	**28**
CUDA4J	5	10	10	**25**

PJ2	10	5	10	**25**
rootbeer	5	8	10	**23**
java-gpu	8	5	10	**23**
JavaCL	8	5	10	**23**
lwjgl	10	10	3	**23**
jacc	5	5	10	**20**
jocl	5	5	10	20
ScalaCL	5	5	10	20

1.3 Nominees

Based on our prior analysis, we will only be focusing on the first two technologies, which obtained the highest grades with 30/30 for Aparapi and 28/30 for JCuda. Those technologies will give us the opportunity to test an openCL and CUDA based solution.

1.4 Battery of tests

Testing our solution will require us to write a first implementation using vanilla Java code, then writing two other implementations - still in Java - but adapted to use a specific solution (in our case, it will be Aparapi and JCuda). We will then be able to compare the implementations.

We will base our tests on two main aspects :

1. Speed
2. Ease of use

1: The first aspect - Speed - will use some benchmarks code testing specific computation aspects like floating point operations or integer manipulations. Those tests will give us precise metrics that will be easily comparable.

2: The second aspect - ease of use - are more kind of subjective and will need a systematic approach so as to be the most fair. We will be looking at some defined criteria, like the number of code lines and the programmer's feedback. The next section will discuse in more details the two aspects.

1.4.1 Speed

To add more relevance to our benchmark, we wanted to choose two benchmarks that does not perform the same type of operations. The most obvious one - a matrix computation - was our first choice. Due to its simple implementation and its ability to be distributed on different cores, we rapidly opted for this one. The matrix multiplication will be able to measure floating point manipulations performance.

Our second choice was to make an array sorting. But, after testing some pieces of code, we went back and looked for another alternative. The idea of the array sorting, as shown in figure 1.1, was to split the array, distribute the subsets and perform a merge sort. This solution was simple to code but performed very badly concerning memory space.

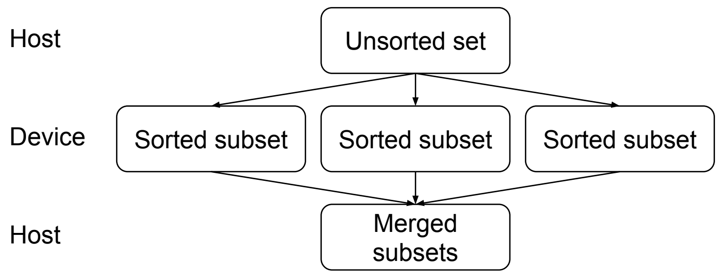


Figure 1.1: Sorting principle

We then looked for the prime factorization of an integer. Again, quite easy to code (see figure 1.2), but this time our implementation wasn't suited to be ported on a multi-threaded architecture and was highly limited by Java integer size, even using *BigInteger*.

We finally opted for the computation of a Levenshtein distance between two strings. The code remains simple, is not heavily restrained by variables' size and can be adapted to a multi-threaded architecture.

To summarize, the speed measure will be based on those two simple computation operations :

1. A matrix multiplication using floating variables

2. Levenshtein distance computation of two strings

1: The matrix multiplication will be as simple as possible, using the

```
BigInteger n = 999999;
BigInteger i = new BigInteger("2");
for(; i.compareTo(n.divide(i)) <= 0; i = i.add(BigInteger.ONE)) {
  while(n.mod(i).equals(BigInteger.ZERO)) {
    System.out.println("Divisor: " + n);
    n = n.divide(i);
  }
}
```

Figure 1.2: Factorization of a number

basic definition saying that each cell of the resulting matrix C=AB is given by the following formula :

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{1.1}$$

This method leads to a complexity of $O(n^3)$ for a squared matrix of size n, which is fine for a benchmark purpose. To have a code as flexible as possible, we won't use any Java matrix objects, but only 2-dimensions doubles arrays. figure 1.3 shows a sample code of the matrix multiplication.

```
  // Computes matrix multiplication C = AB
  public static void matrixMul(double[][] A, double[][] B,
                               double[][] C) {
    for(int i=0; i < size; i++) {
      for(int j=0; j < size; j++) {
        double sum = 0.0;
        for(int k=0; k < size; k++) {
          sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
      }
    }
  }
```

Figure 1.3: Sample code of the matrix multiplication

Adapting the code on a multithreaded architecture won't be difficult since each cell, row or column can be computed independently on a separate core.

The matrix content will be randomized and adjusted according to the matrix size so as to prevent overflows. The size will be set to produce a computation time from 30 seconds to 1 minute. The random seed will be the same among all our tests so as to keep the same conditions.

Figure 1.4 is a quick single measure of the complexity scale of our implementation. The result has been compared to other same measures so as to ensure that there is no edge cases or noise in our results. Raw values of this measure are available in the appendix F.1, and the comparison made with another measure can be found in the appendix G.

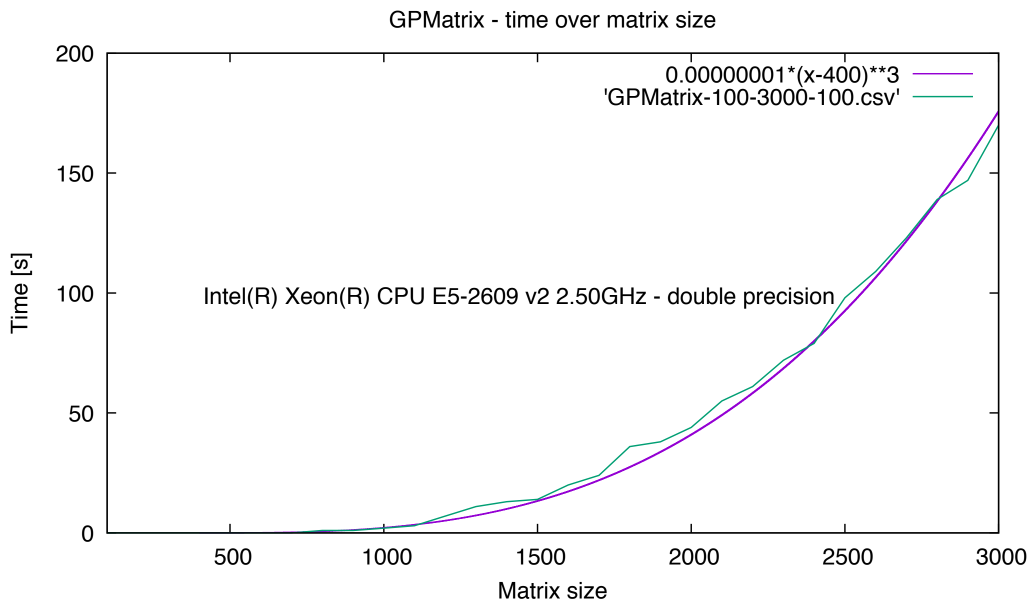


Figure 1.4: Complexity scale of the matrix implementation

2: The Levenshtein distance will be based on the definition found on wikipedia[8] stating that:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if}\min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

We will use an already existing implementation[7] that will be adapted for a multi-threaded use. The strings will be generated randomly and the size will be adjusted to give a computation time from 30 seconds to 1 minute. The strings will be stored in files. It has the advantages of being reusable and not limited by the maximum Java string size or input parameter size. The listing 1.5 is a sample code computing the Levenstein distance. This is the one that will be adapted to be used on the GPU.

```
public static int distance(String a, String b) {
  int [] costs = new int [b.length() + 1];
  for (int j = 0; j < costs.length; j++)
    costs[j] = j;
  for (int i = 1; i <= a.length(); i++) {
    costs[0] = i;
    computeRow(costs, i, a, b);
  }
  return costs[b.length()];
}

// Compute a single row in the distance process
public static void computeRow(int[] costs, int i, String a,
   String b) {
  int nw = i - 1;
  for (int j = 1; j <= b.length(); j++) {
    int cj = Math.min(1 + Math.min(costs[j], costs[j - 1]), a.
        charAt(i - 1) == b.charAt(j - 1) ? nw : nw + 1);
    nw = costs[j];
    costs[j] = cj;
  }
}
```

Figure 1.5: Sample code of the Levenstein distance computation

Figure 1.6 is a measure of the complexity scale of the implementation. We took an average based on 5 executions. In this case, we observe a complexity of $O(n^2)$, which grows much less than the matrix multiplication. Raw values of this measure can be found in the appendix F.2.

1.4.2 Ease of use

To measure the ease of use, we will be looking at :

1. The number of line codes (LOC)

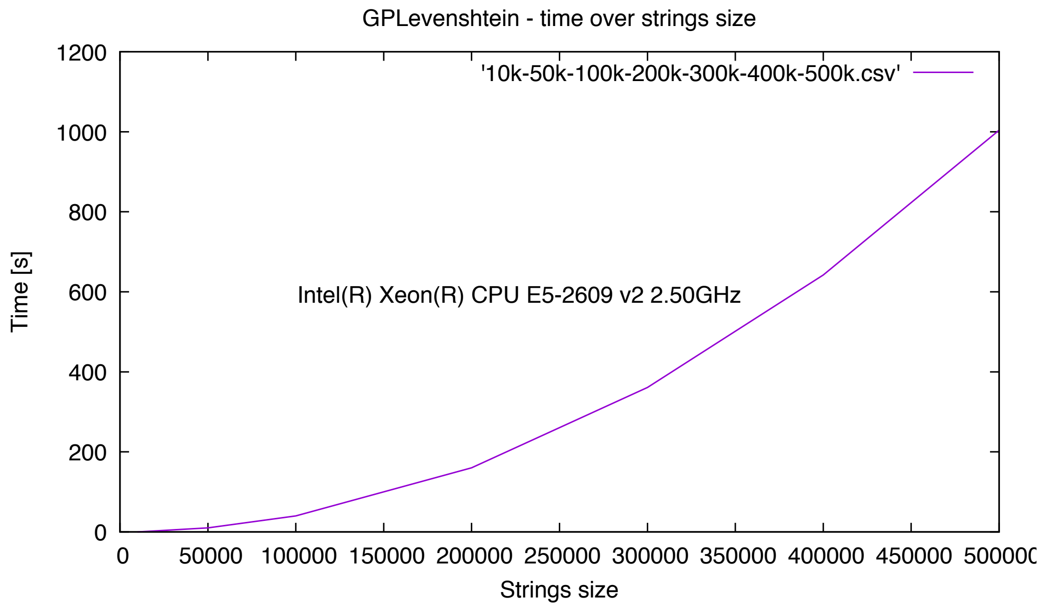


Figure 1.6: Complexity scale of the Levenshtein implementation, average on 5 executions

2. A personal feedback after the solution's installation
3. External reviews of the code by some developers
4. A personal feedback after the execution of the test battery

1: The line of codes will be taken from the routines performing the speed tests mentioned in the previous subsection (1.4.1). Each routine will be evaluated separately and compared to the other implementations. We won't take blank lines and comments into account.

2: The personal feedback will be written right after the setting up of the solution. This feedback will evaluate if the solution is hard to set up or easily usable, like an out-of-the box solution. The environment will already have CUDA and OpenCL set up, so that part will not be evaluated.

3: Having an external review from some developers will help having a fresh point of view. While developing and judging his own code can alter an opinion, someone external will have a neat more objective point of view concerning the complexity of a code. Those persons will be asked to understand the code and give a grade from 1 (very hard) to 5 (very easy) about how it is hard or not to understand how the code uses the GPU.

4: The personal conclusion will be a conclusion after the end of the test battery. It will give an overall feedback about the setting up and development with a particular solution.

1.4.3 Test process

Figure 1.7 shows the sequential process that will be applied for our test battery.

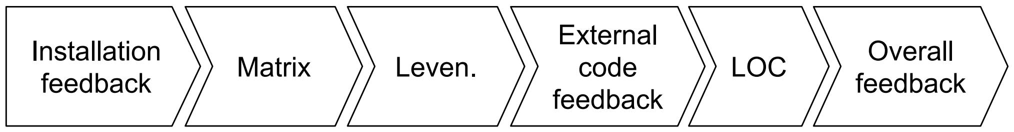


Figure 1.7: Testing process

Chapter 2

Analysis

This chapter will evaluate Aparapi and JCuda - the two candidates who have got the highest marks - based on the test battery discussed in the previous chapter. At the end of this chapter, we will be able to compare the two solutions. This chapter will only be focusing on analyzing the solutions and the comparison between Aparapi and JCuda will come in the next chapter.

2.1 Aparapi

This section will evaluate Aparapi according to the test battery discussed in section 1.4.

Aparapi stands for A PAR{allel} API. It is a tool developed by AMD that gives the ability to write Java code for the GPU. Aparapi works with any configuration supporting a compatible OpenCL 1.1 runtime [3].

To be able to send GPU jobs, Aparapi dynamically translates Java bytecode at runtime to OpenCL instructions [1]. This makes the process of using Aparapi very transparent from the developer's point of view.

2.1.1 Installation

Using Aparapi only requires to include a library jar at the compilation stage and at the execution time. Once done, one can use the Aparapi's API to write specific GPU code.

The jar library comes with several examples that shows how to compile and write code. While we couldn't easily found a getting started guide, but we rapidly understood the process reading some examples.

Aparapi doesn't need any software installation except having a working OpenCL runtime. It only requires to include a jar in the compilation and execution stages and setting up the correct LD_LIBRARY_PATH containing the shared object (.so) library.

The following code shows a typical compilation and run process :

```
$ javac -cp aparapi.jar:. GPLevenshtein.java
$ java -cp aparapi.jar:. GPLevenshtein 1000
```

The next listing shows how we included the shared object library:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/foler
```

The process of using Aparapi is nearly the same as writing CPU threads [5]. Basically, the kernel method will be in a class that inherits from Aparapi's Kernel class. The kernel method is called a given number of times when the class is instantiated. The following example uses Aparapi to perform the addition of two vectors:

```
// The following class contains the kernel method that
// sums two vectors.
// The following code would runs the kernel :
// new AparapiLevenshtein(a,b,result).execute(a.length);
class AparapiExample extends Kernel {
  int[] a;
  int[] b;
  int[] result;

  public AparapiLevenshtein(int[] a, int[] b, int[] result) {
    this.a = a;
    this.b = b;
    this.result = result;
  }

  // Kernel method
  public void run() {
    int i = getGlobalId(); // The thread id
    result[i] = a[i] + b[i];
  }
}
```

2.1.2 Matrix multiplication

The code computing the matrix multiplication with Aparapi can be found in the appendix H.

As seen in figure 2.1, aparapi gives a speedup up to 6 compared to a vanilla execution. It is interesting to note that the speedup is not constant over the matrix size (uncertainty may be the reason, see appendix E), but seems to stagnate around 5. The raw values of the speedup are available in the appendix F.7 and the uncertainty is discussed in appendix E.

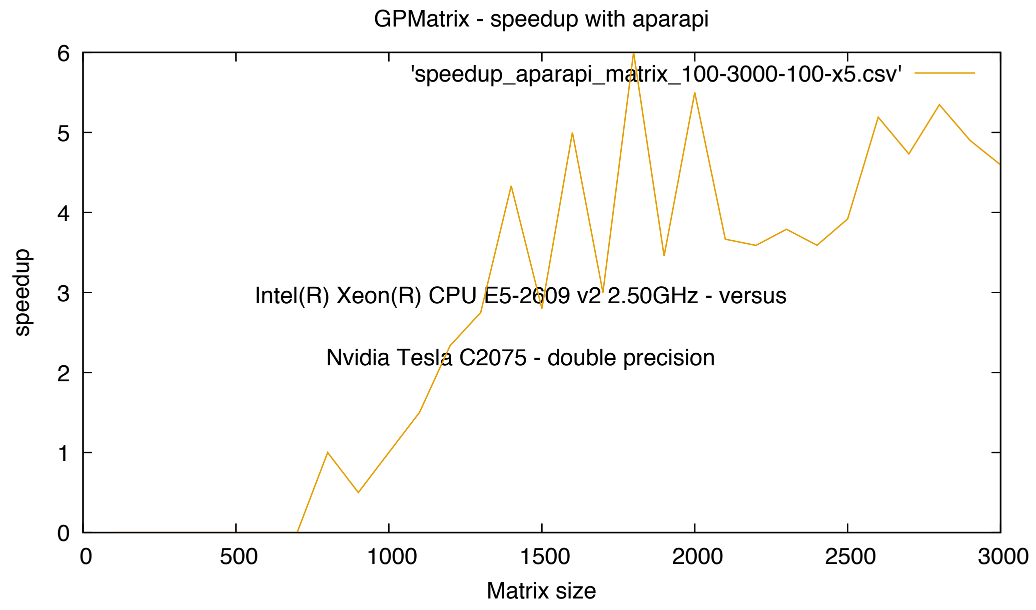


Figure 2.1: Speedup gained using Aparapi on matrix multiplication

Figure 2.2 gives us sequential time running the matrix computation using vanilla Java and Aparapi. For each measure we took the average of 5 executions. We observe that using Aparapi on a GPU doesn't give a smooth curve. Those perturbations could be due to the operating system resource usages during the measures, but we didn't perform any other operations than executing the benchmark and repeated the measures 5 times. It is is probably due the rounding (see appendix E). The raw values of the sequential times using Aparapi on the matrix multiplication are available in the appendix F.3.

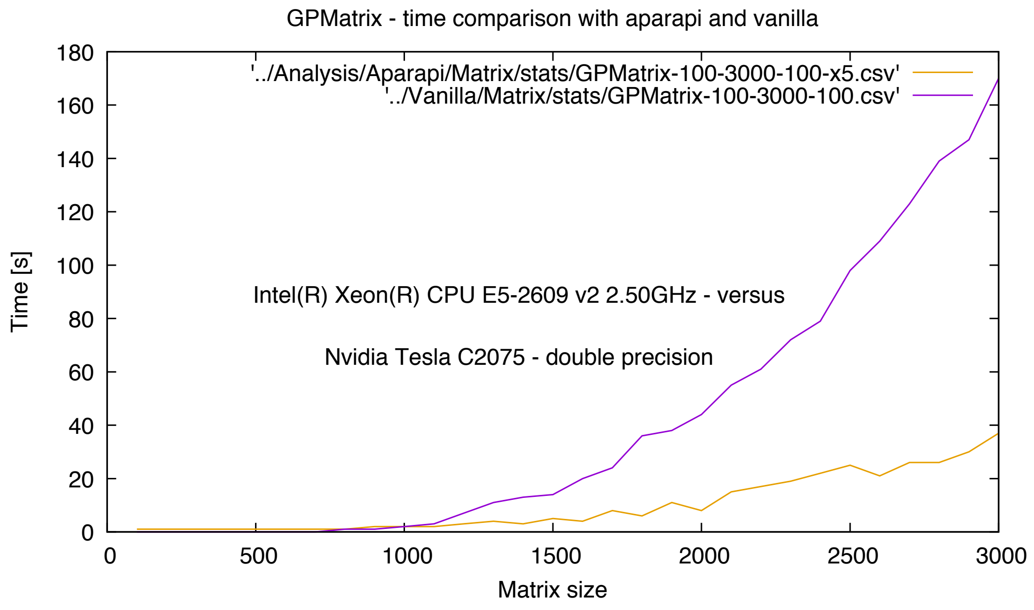


Figure 2.2: Times over matrix size using vanilla Java and Aparapi

Figure 2.3 gives some statistics about the speedup gained with our measures. Those metrics are specifics but, as will the others, it will give us the ability to compare them against another solution - like JCuda.

Speedup average	2,3
Speedup median	3.4
Speedup std. deviation	1,8

Figure 2.3: Speedup statistics of matrix multiplication performed with Aparapi

2.1.3 Levenshtein distance

The code computing the Levenshtein distance with Aparapi can be found in the appendix H.

As seen in figure 2.4, the speedup is at its best with a string size around 200'000 characters. Between this value, the speedup decreases. It is interesting to note that two different types of computation gives a significantly different speedup tendency. In this case, the algorithm is only really efficient around a specific string size range. The raw values of the speedup are available in the appendix F.8 and the uncertainty is discussed in appendix E.

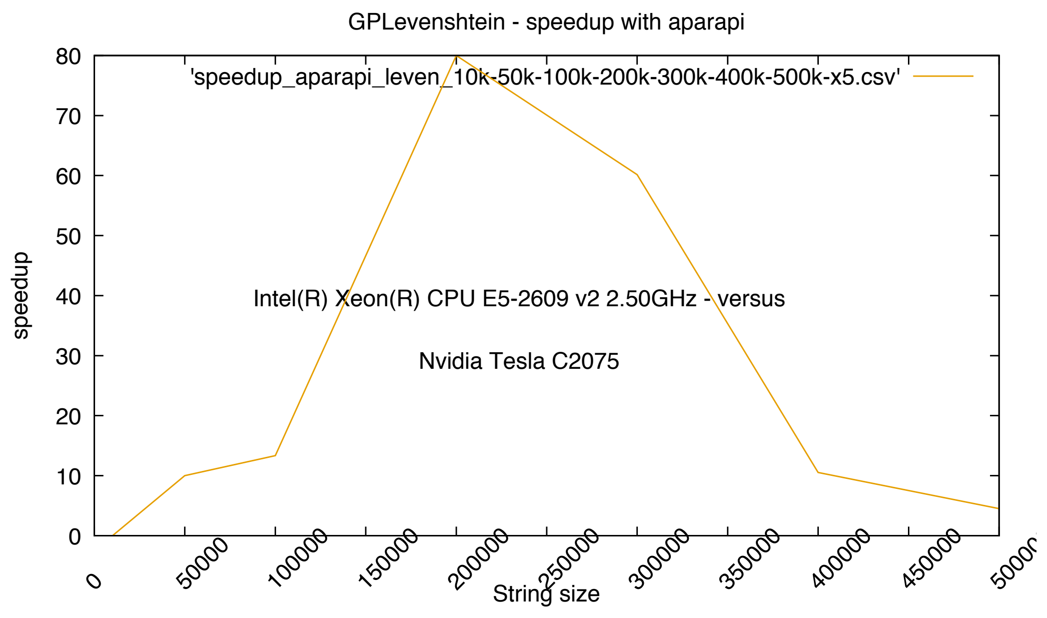


Figure 2.4: Speedup gained using Aparapi on levenstein computation

Figure 2.5 shows sequential time while running levenstein computation on a vanilla implementation and using Aparapi. We can observe that those curves have the same tendency and, around 200'000, show a small decreasing time for the Aparapi execution, thus giving the kind of 'mountain' for the speedup. Raw values of the sequential time using Aparapi for the Levenshtein distance computation is available in the appendix F.4.

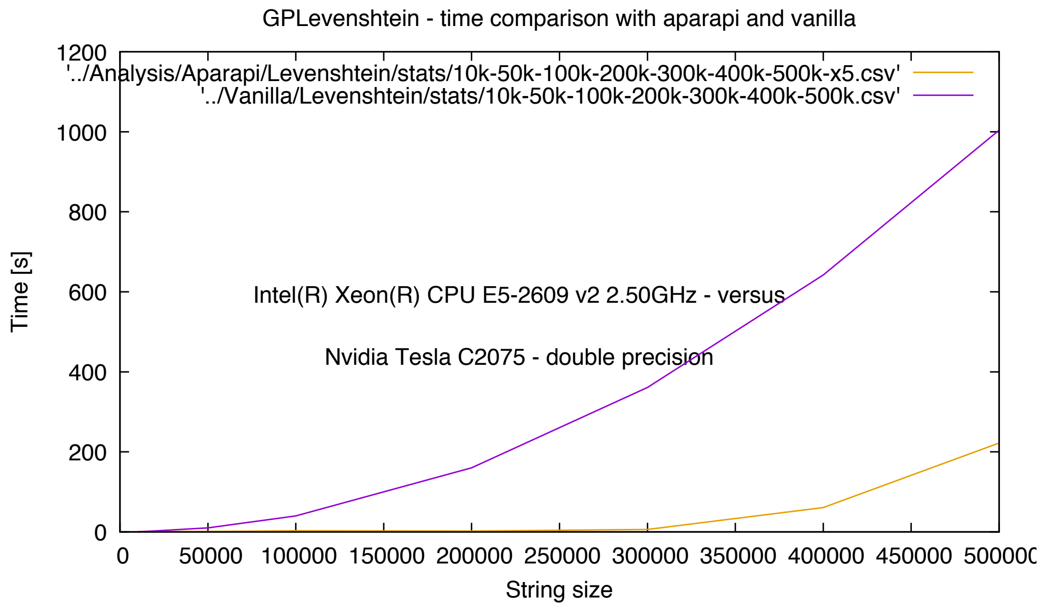


Figure 2.5: Times over string sizes using java and Aparapi

Figure 2.6 shows some statistics of the speedup gained with our measures. As observed in figure 2.4, we come with a high standard derivation, due to the kind of 'mountain' we obtained in our graph.

Speedup average	25,2
Speedup median	10
Speedup std. deviation	29

Figure 2.6: Speedup statistics of levenstein computation performed with Aparapi

2.1.4 External code feedback

We asked 2 developers to evaluate the difficulty of using Aparapi to write GPU code. We gave them the Matrix multiplication and the Levenshtein distance source codes asking them to give a mark from 1 (very difficult to understand) to 5 (very easy to understand). The surveys can be found in the appendix H.

Results are given in figure 2.7:

	Matrix	**Levenstein**
Developper 1	5	5
Developper 2	5	5

Figure 2.7: Results of the Aparapi surveys. From 1 (very difficult to understand) to 5 (very easy to understand)

2.1.5 LOC

The number of code lines needed to use Aparapi for the matrix multiplication and Levenshtein distance computation are given in figure 2.8. Those numbers only include the methods used to compute the matrix multiplication and the Levenstein distance, removing comments and blank lines.

	vanilla	**Aparapi**	**delta**
GPMatrix	13	21	8
GPLevenstein	18	24	6

Figure 2.8: LOC of GPMatrix and GPLevenstein using Aparapi compared to vanilla

2.1.6 Overall feedback

Installing and using Aparapi is quite straightforward. The first problem we met was to find an official documentation about installing and using Aparapi. While we couldn't find any relevant official documentation, we came up using a given example to take our first steps. Reading an example gave us enough information to start adapting our code using Aparapi.

The second problem we met was that using Java object inside the kernel function is not possible. We had to remove all Java object usages and adapt our implementation - for example by using a char array instead of a string in GPLevenstein. This restriction wasn't clearly documented by Aparapi.

The third problem we met was that getting the length of an array raised an error. After investigating, we found out that it is a know bug [2]. This step took us a while to discover since the raised error wasn't explicit and didn't directly guide us to the real error's source.

In conclusion, Aparapi was quite easy to use despite the loss of an official and complete documentation. Not having the ability to use Java objects is disappointing and removes a major advantage of using Java code. But for specific uses, Aparapi is a simple and efficient solution to perform GPU computation inside Java code.

2.2 JCuda

This section will evaluate JCuda according to the test battery discussed in section 1.4.

JCuda is a binding library that uses NVIDIA CUDA technologies. CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)[6].

JCuda provides binding from Java to CUDA. The library enables programmers to write Java methods. Thoses methods will be automatically bound to the corresponding CUDA ones, which would be written in CUDAC/C++ [4].

With JCuda, programmers will have to write the kernel method in a separate native CUDA file (.cu) and use the JCuda library from Java to call the kernel and perform data transfers. The process is nearly similar as one would do with plain CUDA language, but using adapted Java methods.

2.2.1 Installation

The installation of JCuda requires the same process as with Aparapi. A jar library needs to be included in the compilation and execution stages and a library (shared object file on Linux) must be visible to Java at runtime (typically using LC_LIBRARY_PATH).

No extra software is needed, but only having a working CUDA environment setup and including the JCuda library.

The installation process is well documented as well as the first steps to write a first program using JCuda. There are also plenty of examples to help understanding how JCuda works.

A typical compilation and run case for a JCuda program would be something like :

```
$ javac -cp ".:jcuda-0.7.5.jar" GPMatrix.java
$ java  -cp ".:jcuda-0.7.5.jar" GPMatrix 1000
```

To get an idea of how JCuda works, the following listing shows an example of a two vectors addition using JCuda :

```
// From the Java class
  ...
  int[] A = ...; int[] B = ...;
  int[] result = ...; int size = a.length;

  // Create the PTX file by calling the NVCC
  String ptxFileName = preparePtxFile("kernel.ptx");

  // Load the CUDA kernel ptx file
  CUmodule module = new CUmodule();
  cuModuleLoad(module, ptxFileName);

  // Obtain a function pointer to the kernel function
  CUfunction function = new CUfunction();
  cuModuleGetFunction(function, module, "add");

  // Allocate the device input/output data, and
  // copy the host input data to the device
  int ptrSize = size * Sizeof.INT;
  CUdeviceptr deviceInputA = new CUdeviceptr();
  cuMemAlloc(deviceInputA, ptrSize);
  cuMemcpyHtoD(deviceInputA, Pointer.to(A), ptrSize);

  CUdeviceptr deviceInputB = new CUdeviceptr();
  cuMemAlloc(deviceInputB, ptrSize);
  cuMemcpyHtoD(deviceInputB, Pointer.to(B), ptrSize);

  CUdeviceptr deviceOutput = new CUdeviceptr();
  cuMemAlloc(deviceOutput, ptrSize);

  // Set up the kernel parameters: A pointer to an array
  // of pointers which points to the actual values
  Pointer kernelParameters = Pointer.to(
    Pointer.to(deviceInputA),
    Pointer.to(deviceInputB),
    Pointer.to(deviceOutput),
    Pointer.to(new int[]{size})
  );
  // Kernel call parameters
  int blockSizeX = 256;
  int gridSizeX = (int)Math.ceil((double)size / blockSizeX);

  // ————————————————————————————————
  long startTime = System.currentTimeMillis();
  // --- Start of benchmark zone --->
  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
  );
  cuCtxSynchronize();

  // Get the result
  cuMemcpyDtoH(Pointer.to(result), deviceOutput, ptrSize);
...

// Kernel.cu
extern "C"
__global__ void add(int* A, int* B, int* result, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < size) {
    result[i] = A[i] + A[i];
  }
}
```

2.2.2 Matrix multiplication

Like we did with Aparapi, we took the matrix multiplication vanilla code and ported the same portion of code as with Aparapi to the GPU. We then made the same measures as with Aparapi and computed the speedup gained between JCuda and vanilla Java code. The adapted code can be found in the appendix I.

Figure 2.9 shows the speedup gained using JCuda in the matrix multiplication code. We can see that we obtained a speedup rising up to 12. The shape of the curve is not constantly rising, but is giving a maximum around a matrix size between 2000 and 2500.

So as to prevent edge cases and removing noise, we each time took 5 measures and used the average values between the 5 results. Raw values for the speedup using JCuda on the matrix multiplication are available in the appendix F.9 and the uncertainty is discussed in appendix E.

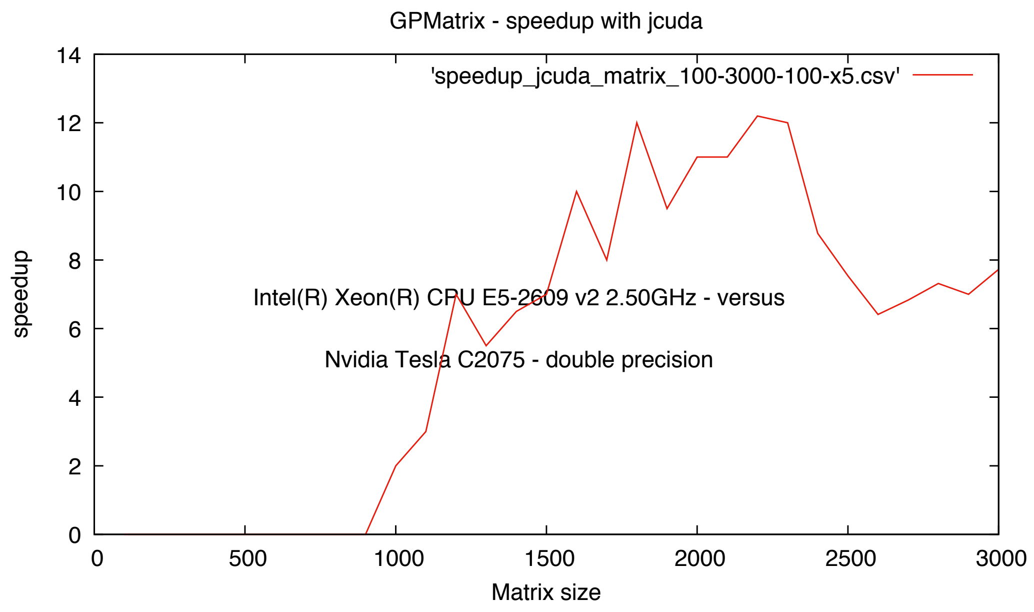


Figure 2.9: Speedup gained using JCuda on matrix multiplication

Figure 2.10 gives us sequential times of the vanilla execution and the JCuda one on the matrix multiplication. The sequential times with JCuda seem to expand on a logarithm form while the sequential times on the vanilla code expand more in an exponential manner. Raw values of the sequential times using JCuda on the matrix multiplication can be found in the appendix F.5.

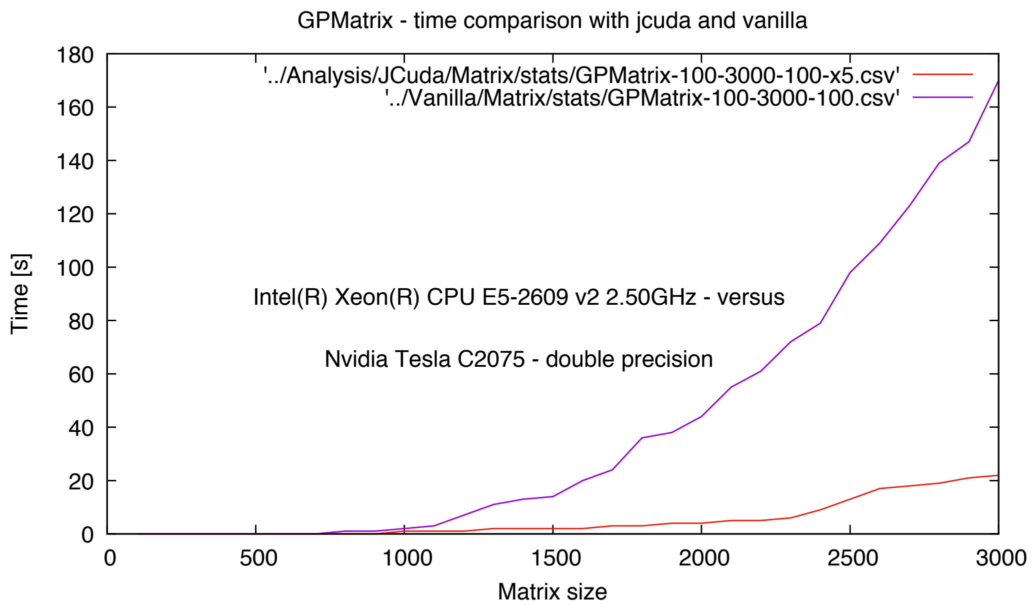


Figure 2.10: Times over matrix size using Java and JCuda

Figure 2.11 are some statistics on the speedup gained using JCuda. Those values will later be used to have metrics to compare between JCuda and Aparapi.

Speedup average	5.6
Speedup median	7
Speedup std. deviation	4.3

Figure 2.11: Speedup statistics of matrix multiplication performed with JCuda

2.2.3 Levenshtein distance

Figure 2.12 is the speedup gained using JCuda for the Levenshtein distance against vanilla code. The speedup gives its highest peak around a string size of 200'000 and decreases after. We observed the same behavior using Aparapi. Measures were based on an average of 5 executions and raw values are available in appendix F.10. The uncertainty is discussed in appendix E. Full code can also be found in the appendix I.

Unlike with the matrix multiplication, Aparapi gave a better peak with a speedup of nearly 80, while JCuda is around 55 at its highest peak.

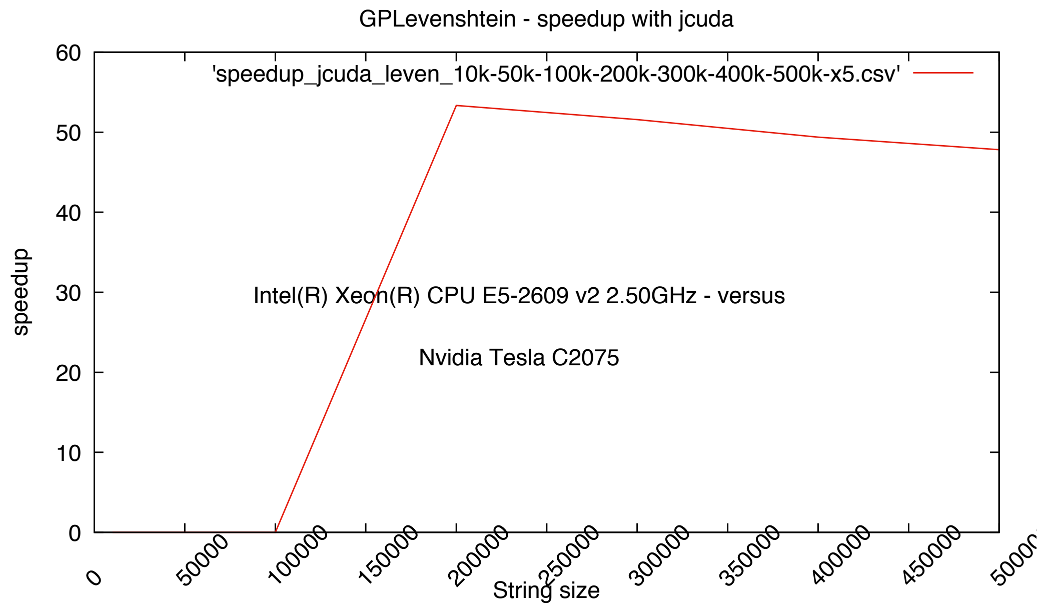


Figure 2.12: Speedup gained using JCuda on Levenstein distance computation

Figure 2.13 shows sequential times while executing the Levenshtein distance computation on different string sizes with JCuda and vanilla Java, this on the same graph. Sequential times for JCuda are ridiculously small compared to the vanilla Java. Those measures are based on an average of 5 executions. Raw values of the execution times for the Levenstein distance computation using JCuda are available in appendix F.6.

Statistics about the speedup are available in figure 2.14.

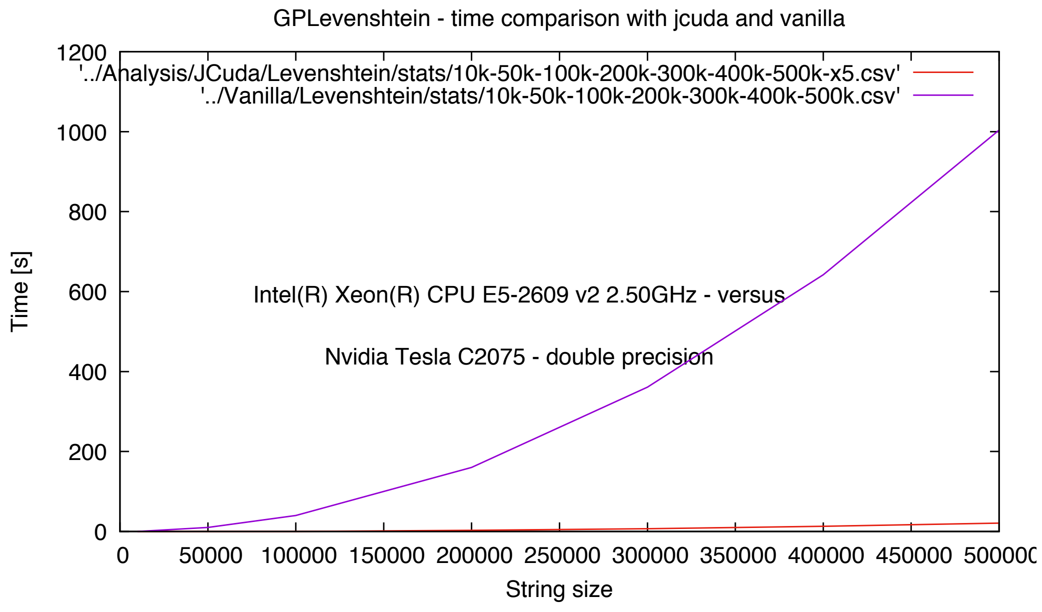


Figure 2.13: Times over string size using Java and JCuda

Speedup average	28.5
Speedup median	47.8
Speedup std. deviation	24.8

Figure 2.14: Speedup statistics of Levenstein computation performed with JCuda

2.2.4 External feedback

We asked 2 developers to evaluate the difficulty of using JCuda to write GPU code. We gave them the Matrix multiplication and the Levenshtein distance source codes, asking them to give a mark from 1 (very difficult to understand) to 5 (very easy to understand). The surveys can be found in the appendix I.

Results are given in the figure 2.15:

	Matrix	**Levenstein**
Developper 1	4	3
Developper 2	4	3

Figure 2.15: Results of the JCuda surveys. From 1 (very difficult to understand) to 5 (very easy to understand)

2.2.5 LOC

The numbers of code lines needed to use JCuda for the matrix multiplication and Levenshtein distance computation are given in figure 2.16. Those numbers include only the methods used to compute the matrix multiplication and the Levenshtein distance, removing comments and blank lines.

	vanilla	**JCuda**	**delta**
GPMatrix	13	47	30
GPLevenstein	18	51c	25

Figure 2.16: LOC of GPMatrix and GPLevenstein using JCuda compared with vanilla

2.2.6 Overall feedback

JCuda is really well documented and easy to install. Due to its binding philosophy, using JCuda gives more the impression of writing CUDA code in Java than using Java to write GPU code. The fact that the kernel method is written in C is the main reason, but also the way we allocate and set the kernel's parameters. From Java, all we do is calling the kernel exactly as we would with CUDA, but with Java methods. This can be disturbing for someone who has no experience using CUDA, but on the other hand, if someone has already a CUDA experience, it won't be difficult at all to get started.

A major restriction of using JCuda is due to the kernel written in C. It does not allow developers to use Java code and breaks the process of a pure Java development workflow. This lets only the possibility to write a single c method. In our case, for example, we couldn't use or write a simple method **min()** that would make the code much easier to read in our kernel. But, in the other hand, an advantage of using the kernel in its pure form, written in C, is the fact that it makes possible to use already written CUDA kernel methods. Hence making the process of kernel reusability among other existing projects possible. This aspect should be worth considering in some context.

In conclusion, JCuda was easy to install, well documented and relatively fast to get started with. Using pure kernel methods in C could be interesting for someone familiar with CUDA, but does not have the advantage of using Java code. For Java code that needs to execute specific instructions to the GPU on a CUDA architecture, JCuda is a good solution as long as the kernel does not need any Java objects and/or methods.

Chapter 3

Results

This chapter will finally compare Aparapi and JCuda based on the test battery we applied on them in the previous chapter.

As discussed in the introduction (see section 1.1), our comparison is based on the execution speed and ease of use. We will discuss each comparison point for Aparapi and JCuda so as to evaluate which one is better for each criteria.

The next chapter will be an overall conclusion, bringing the final thoughts and perspectives.

3.1 Execution speed

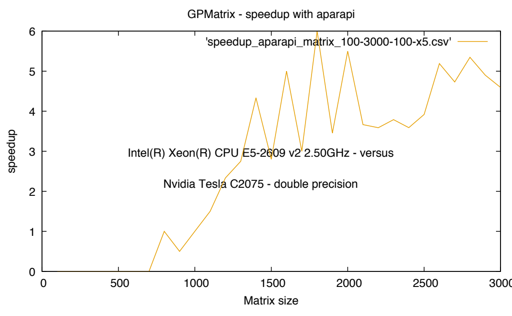
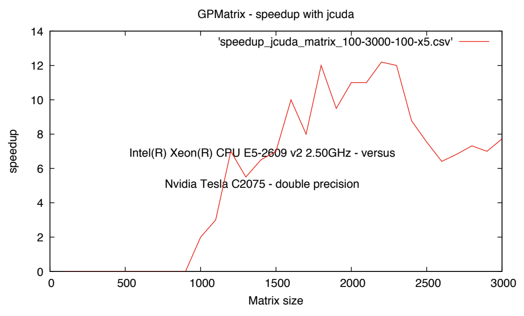
For the execution speed, two benchmarks were used and executed against Aparapi and JCuda. The baseline used for the speedup were a native execution on a vanilla Java code using only the host's CPU.

Each benchmark is testing a different type of computation. The matrix multiplication is more focused on floating point operations while the Levenshtein distance is more focused on char operations.

For each benchmark, we will compare the speedup gained using Aparapi and JCuda.

3.1.1 Matrix

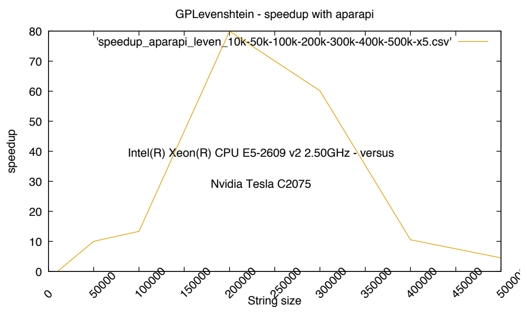
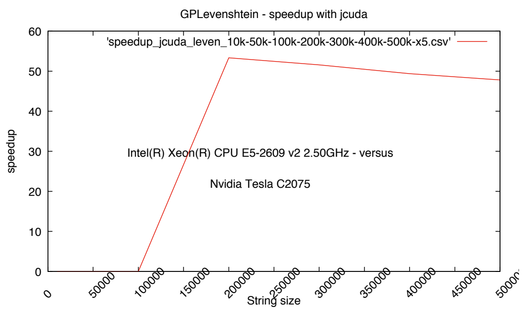
The following table is a side by side comparison of Aparapi and JCuda for the matrix multiplication, based on the speedup.

Aparapi	**JCuda**
Speedup plot	
	
Speedup peak	
6	**12.2**
Speedup average	
2.3	**5.6**
Speedup median	
3.4	**7**

On this turn, JCuda shows a clear better performance on all the metrics, having globally an execution time 2 times faster than Aparapi. But it is worth noting that JCuda seems to suddenly decrease performance after a certain matrix size (around 2400), while Aparapi show a globally more stable speedup. But even with this consideration, JCuda always shows a better speedp.

3.1.2 Levenshtein

The following table is a side by side comparison of Aparapi and JCuda for the Levenstein distance computation, based on the speedup.

Aparapi	**JCuda**
Speedup plot	
	
Speedup peak	
80	53.3
Speedup average	
25.2	**28.5**
Speedup median	
10	**47.8**

Unlike with the matrix multiplication, this time we can't clearly define which solution is faster. Aparapi is globally less fast than JCuda but has the best peak with up to 80 times faster compared with JCuda - who has up to 53.3 times faster. The only answer we can give is "it depends". Around a string size of 200k, Aparapi is better than JCuda. But as soon as we have a bigger string size (around 300k), JCuda is a faster solution and shows a more stable speedup with an increasing string size. Based on the average speedup, JCuda is globally faster.

3.2 Ease of use

To evaluate the ease of use, we will be looking at the following aspects:

1. The installation process
2. The number of code lines
3. The external feedbacks
4. The problems encountered and restrictions

3.2.1 Installation process

In both cases - with Aparapi and JCuda - the installation process were very similar and fast. Despite the fact that we didn't have any difficulties getting started and installing both of the solutions, we give the medal to JCuda for this turn.

JCuda was easier to instal due to a better documentation, providing examples and command lines to compile and execute a JCuda based program. This support wasn't as good with Aparapi, where we didn't really find any good getting started explanation.

3.2.2 Line of codes

The following table is a side by side comparison of Aparapi and JCuda concerning the line of codes needed to run the matrix multiplication and Levenshtein distance computation. In parenthesis are the differences with the number of code lines for the vanilla Java implementation.

Aparapi	**JCuda**	**(Vanilla)**
	Matrix multiplication	
21 (+8)	47 (+34)	13
	Levenstein distance computation	
24 (+6)	51 (+38)	18

Aparapi wins by far this turn with more than 2 times less lines of code needed to perform the same task than with JCuda.

Aparapi was much easier to implement in terms of code lines needed. This difference with JCuda is mainly due to the need for JCuda to set each kernel attribute so it is available to the kernel method - which is written in C. This process requires many steps. For a quick use, Aparapi is more suitable than JCuda.

3.2.3 External feedback

The following table summarizes the external feedback on how developers evaluated the difficulty to understand the code - especially the part performing computation on the GPU. They had to give a mark from 1 (very difficult) to 5 (very easy). In the following table are the averages' marks.

Aparapi	**JCuda**
Matrix multiplication - external developers feedback average marks	
5	3.5
Levenstein distance - external developers average feedback marks	
5	3.5

Aparapi seems to be more easier to understand than JCuda. It is probably due to the fact that JCuda requires more lines of code and a kernel written in C. The external feedback were performed with 2 developers having already an experience in GPU programming.

3.2.4 Problems encountered - restrictions

Aparapi	**JCuda**
• No official getting started guide found. We had to use examples so as to start using Aparapi. • Impossible to use any Java objects inside the kernel method. We had to write our own min() methods and convert strings to char arrays. • Impossible to get the size of an array inside the kernel. It is a known issue [2].	• Like Aparapi, impossible to use Java objects but even any Java code inside the kernel method. • The kernel method is written in C, not in Java. This restricts the use of Java but makes the reusability and sharing of existing CUDA kernel methods possible. • Impossible to write a simple min() method for the kernel, making the code hard to read.

In terms of restrictions, we can't clearly give a single medal to Aparapi or JCuda. The answer is "it depends". Either Aparapi or JCuda can't use Java objects in the kernel methods. Aparapi has the advantage of being fully usable under a Java workflow, while JCuda needs a kernel written in C. Aparapi gives the ability to have custom methods available from the kernel, giving more flexibility. On the other hand, JCuda has it's kernel written in C, giving the ability to use already written CUDA kernel methods. This also makes the use of JCuda from someone already knowing CUDA easy, since the kernel code remains the same, and the process, for JCuda, of calling the kernel method is also very similar as one we would do with vanilla CUDA programming.

For someone already knowing CUDA, JCuda gives a better way to use the GPU from a Java code. But this only under a CUDA compatible environment.

For someone looking for a full Java solution to write GPU code, having an OpenCL environement and giving the ability to write flexible methods available from the kernel, Aparapi is a better solution.

Chapter 4

Conclusion

This chapter is the final words that will conclude our analysis of Aparapi and JCuda.

4.1 Aparapi or JCuda?

For someone absolutely looking for the fastest solution, JCuda is more suitable than Aparapi. In our analysis, based on a matrix multiplication and a Levenshtein distance, JCuda is globally faster than Aparapi, with sometimes results more than 2 times faster than Aparapi. However, we observed one case on a particular string size for the Levenshtein distance computation where Aparapi was faster. So, even speaking of speed, Aparapi could be faster on particular cases. Globally, JCuda is faster but we couldn't test every cases on every computation types.

Speaking about ease of use, Aparapi wins the palm and is more suitable than JCuda. It requires less line of codes, has the advantage of being entirely usable in Java and doesn't need an CUDA environment, which is not always possible for someone not having an NVDIA GPU.

JCuda requires a kernel written in C, which could be either an advantage or a disadvantage. Someone already familiar with JCuda won't have any difficulties since the kernel whould be the same, but for someone not familiar with CUDA programming it will be more difficult to get started compared to Aparapi.

If keeping an absolute Java environment is a priority, Aparapi comes as the only possible solution. With Aparapi, everything is written in Java, providing a smooth development workflow and a clear code. It also gives the abilities of writing custom Java methods helper available from the kernel - which is not possible with JCuda.

4.2 The answer

Answering the question "should I use Aparapi od JCuda?" requires to know what are the needs and the development environment available. It maily requires to answer three questions :

1. Do we have a CUDA or OpenCL environment?
2. Do I want to keep the code simple and stay exclusively with Java?
3. Am I focused on speed?

The process to guide the choice of Aparapi or JCuda based on the previous questions is given by the flowchart in the figure 4.1.

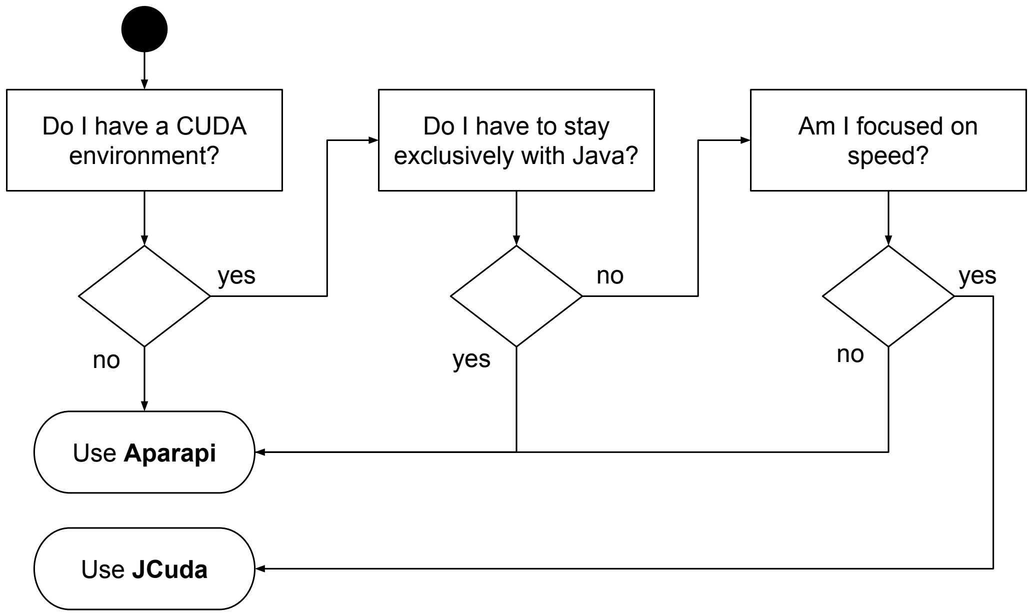

Do I have a CUDA environment?
yes
no
Do I have to stay exclusively with Java?
no
yes
Am I focused on speed?
yes
no
Use Aparapi
Use JCuda

Figure 4.1: Flowchart guiding the decision of using either Aparapi or JCuda

However, this process may not be true on every cases, especially speaking of speed where Aparapi could be faster on some cases - and we obviously couldn't test every cases for every computation types. We are only saying that JCuda is globally faster, but there could be cases where it is not. The last question on the flow "Am I focused on speed?" could lead to JCuda even if the answer is "no" for someone who prefers using a CUDA based solution anyway. At this point of the flow, it is more a question of flavor. We tried, on the flow, to fit the most global use case and answer the question for, what we think, corresponds to the majority.

4.3 Perspectives

There are some points that could be expanded and leave some future perspectives. We are thinking of :

- Testing more solutions
- Developping more benchmarks
- Increasing the measures' precisions

Aparapi and JCuda are of course not the only solutions that make GPU programming available from Java. In our pre-analysis (see table 1.2.4), we had 11 candidates and we only evaluated 2 of them. We chose the first 2 of them that seemed the most promising to us, but evaluating more of them according to our test battery could lead us to better solutions.

Our test battery only uses 2 benchmarks. Having more benchmarks would give us more specific measures about what solution is the best for what kind of computation and parameter. As we saw, JCuda is globally faster than Aparapi. But in special cases, Aparapi is faster. With more benchmarks we would perhaps discover more special cases where Aparapi is faster, and then giving a more precise way to make a decision between Aparapi and JCuda.

Our benchmark only measures the rounded seconds spent. Hense, the sequential time and the speedup suffer from an uncertainty that could be important in some cases. The uncertainty in our measures is discussed in appendix E. The uncertainty could be much reduced by measuring the sequential time in milliseconds instead of rounded seconds.

Bibliography

[1] AMD. *AMD dev blog.* [Online; accessed 11-mai-2016]. 2011. URL: `http://developer.amd.com/community/blog/2011/09/14/i`.

[2] AMD. *Aparapi repo issues.* [Online; accessed 11-mai-2016]. 2016. URL: `https://github.com/steelted/aparapi/issues/117`.

[3] Aparapi. *Aparapi repos.* [Online; accessed 4-mai-2016]. 2016. URL: `https://github.com/aparapi/aparapi/blob/master/doc/FrequentlyAskedQuestions.md`.

[4] Jitendra Parande Jayshree Ghorpade. "GPGPU PROCESSING IN CUDA ARCHITECTURE". In: *arxiv* (2012).

[5] Shrinivas Joshi. "Leveraging Aparapi to Help Improve Financial Java Application Performance". In: *AMD-dev* (2012).

[6] NVIDIA. *CUDA website.* [Online; accessed 4-mai-2016]. 2016. URL: `http://www.nvidia.com/object/cuda_home_new.html`.

[7] rosettacode. *Java levenshtein.* [Online; accessed 4-April-2016]. 2016. URL: `https://rosettacode.org/wiki/Levenshtein_distance#Java`.

[8] Wikipedia. *Levenshtein.* [Online; accessed 4-April-2016]. 2016. URL: `https://en.wikipedia.org/wiki/Levenshtein_distance`.

Declaration of Authorship

I hereby certify that the paper I am submitting is entirely my own original work except where otherwise indicated. I am aware of the regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Place	Date	Signature

Appendix A

Source code structure

Official repository address: https://github.com/nkcr/JavaGPU

This is the list of the four root folders and their contents :

Analysis	Code and measures using different Java GPU technologies
Common	Common stuff needed across the project like libraries
Results	Contains comparison measures like speedup
Vanilla	Code written in pure Java so as to have base metrics

Appendix B

Versions used

Java	1.7.0_95
CUDA	7.5.18
OpenCL	Platform: NVIDIA CUDA, version 1.1
JCuda	0.7.5 for CUDA 7.5.18
Aparapi	1.0.0 linux x86_64
Ubuntu	Ubuntu 14.04.4 LTS
Bash	RELEASE: 2.1
GNUplot	Version 4.6 patchlevel 4

Appendix C

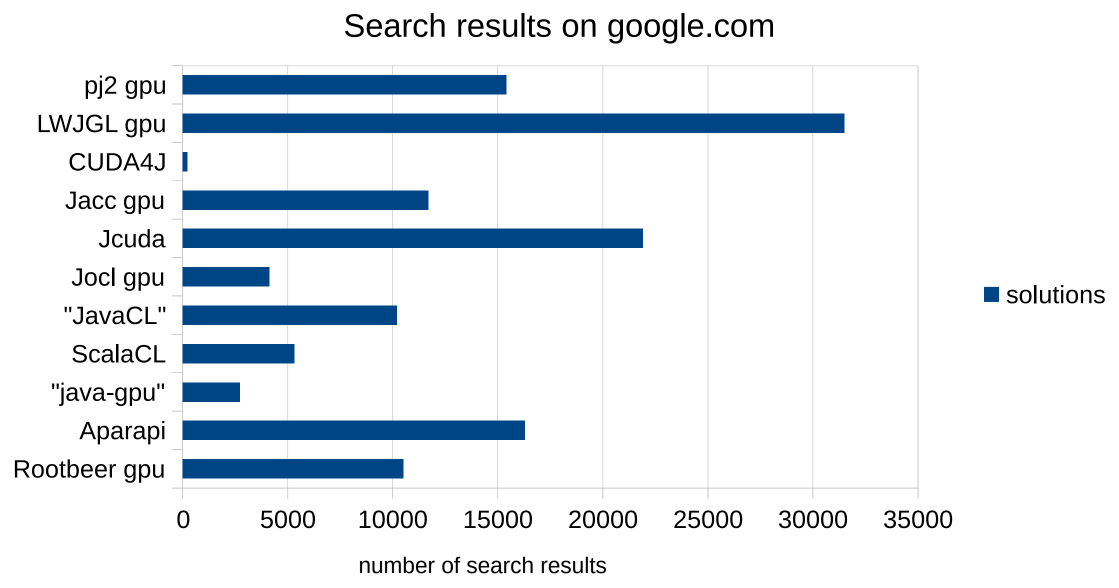
List of initial candidates

Liste des candidats[1]

Nom	Description	Lien
	(Byte)code translation and OpenCL code generation	
aparapi	An open-source library that is created and actively maintained by AMD. In a special "Kernel" class, one can override a specific method which should be executed in parallel. The byte code of this method is loaded at runtime using an own bytecode reader. The code is translated into OpenCL code, which is then compiled using the OpenCL compiler. The result can then be executed on the OpenCL device, which may be a GPU or a CPU. If the compilation into OpenCL is not possible (or no OpenCL is available), the code will still be executed in parallel, using a Thread Pool.	https://github.com/aparapi/aparapi
rootbeer1	An open-source library for converting parts of Java into CUDA programs. It offers dedicated interfaces that may be implemented to indicate that a certain class should be executed on the GPU. In contrast to Aparapi, it tries to automatically serialize the "relevant" data (that is, the complete relevant part of the object graph!) into a representation that is suitable for the GPU.	https://github.com/pcpratts/rootbeer1
java-gpu	A library for translating annotated Java code (with some limitations) into CUDA code, which is then compiled into a library that executes the code on the GPU. The Library was developed in the context of a PhD thesis, which contains profound background information about the translation process.	http://code.google.com/p/java-gpu/
ScalaCL	Scala bindings for OpenCL. Allows special Scala collections to be processed in parallel with OpenCL. The functions that are called on the elements of the collections can be usual Scala functions (with some limitations) which are then translated into OpenCL kernels.	https://github.com/ochafik/ScalaCL
	Language extensionsJava OpenCL/CUDA binding libraries	
JavaCL	A language extension for Java that allows parallel constructs (e.g. parallel for loops, OpenMP style) which are then executed on the GPU with OpenCL. Unfortunately, this very promising project is no longer maintained.Java bindings for OpenCL: An object-oriented OpenCL library, based on auto-generated low-level bindings.	https://github.com/ochafik/JavaCL
jocl	Java bindings for OpenCL: An object-oriented OpenCL library, based on auto-generated low-level bindings.	http://jogamp.org/jocl/www/
lwjgl	Java bindings for OpenCL: Auto-generated low-level bindings and object-oriented convenience classes.	http://www.lwjgl.org/
jocl	Java bindings for OpenCL: Low-level bindings that are a 1:1 mapping of the original OpenCL API.	http://jocl.org/
jcuda	Java bindings for CUDA: Low-level bindings that are a 1:1 mapping of the original CUDA API.	http://jcuda.org/
Jacc	Java frameworks that performs on low-level at the runtime. Experimental phase.	http://arxiv.org/pdf/1508.06791.pdf
CUDA4J	Java API made by IBM that gives many classes for GPU computing via CUDA peripheral.	https://www-01.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm

1 Partially taken from http://stackoverflow.com/questions/22866901/using-java-with-nvidia-gpus-cuda/22868938#22868938 – edited Mar 17'15

Nom	Description	Lien
		.java.lnx.71.doc/user/gpu_developing_cuda4j.html
Com.ibm.gpu	Another java api made by ibm to sort primitive arrays via gpu.	https://www-01.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.lnx.71.doc/user/gpu_developing_sort.html
lwjgl	Lightweightjava game library. A game library that gives the ability to use OpenCL for parallel programming.	https://www.lwjgl.org/
PJ2	Parallel Java 2 (PJ2) is an API and middleware for parallel programming in 100% Java on multicore parallel computers, cluster parallel computers, hybrid multicore cluster parallel computers, and GPU accelerated parallel computers[2].	https://www.cs.rit.edu/~ark/pj2.shtml



2 https://www.cs.rit.edu/~ark/pj2.shtml

Appendix D

Installation of CUDA in Ubuntu 14.04

```
$ wget http://developer.download.nvidia.com/compute/cuda/repos
    /ubuntu1404/x86_64/cuda-repo-ubuntu1404_7.5-18_amd64.deb
$ sudo apt-get update
$ sudo apt-get install cuda
$ export CUDA_HOME=/usr/local/cuda-7.5
$ export LD_LIBRARY_PATH=${CUDA_HOME}/lib64

$ PATH=${CUDA_HOME}/bin:${PATH}
$ export PATH

$ wget http://us.download.nvidia.com/XFree86/Linux-x86_64
    /352.79/NVIDIA-Linux-x86_64-352.79.run
$ chmod +x NVIDIA-Linux-x86_64-352.79.run
$ sudo ./NVIDIA-Linux-x86_64-352.79.run

$ cuda-install-samples-7.5.sh ~/cuda
$ cd ~cuda/NVIDIA_CUDA-7.5_Samples
$ cd 1_Utilities/deviceQuery
$ make
```

For the driver to be installed, Xserver must be disabled :

```
# /etc/default/grub
$ GRUB_CMDLINE_LINUX_DEFAULT="text"
```

Appendix E

Uncertainty discussed

This appendix is a post-reflection on how the uncertainty impacts our results.

All our measures were made in rounded seconds, this means that all our data suffer from an uncertainty that could impact the results and the conclusions made. We will compute the uncertainty and re-plot the speedups that were used in this report, but this time showing the uncertainty.

First, let's find the formula giving the uncertainty. For a single measured value, say 10 seconds, we know that is has been rounded. It could be in reality 10.1 seconds, 10.5 seconds or, the maximum, 10.999999 seconds. Thus, the measure could be in reality from 10 seconds to 10.9999 seconds. Therefor, we have an uncertainty of maximum $+1$ seconds.

Based on this principle, we computed the uncertainty of the speedup (S) based on two measures (x_1, x_2) :

$$S = \frac{x_1}{x_2}$$

$$S_{max} = \frac{x_1 + 1}{x_2}$$

$$S_{min} = \frac{x_1}{x_2 + 1}$$

$$S_{uncertainty} = S_{max} - S_{min} = \frac{x_1 + 1}{x_2} - \frac{x_1}{x_2 + 1} = \frac{x_1 + x_2 + 1}{x_2^2 + x_2}$$

Given the previous formula, we re-plotted the speedup with the uncertainty. Figures E.1 and E.2 show speedups of Aparapi and JCuda on the matrix multiplication and Levenshtein distance respectively with the uncertainty. We can see that the uncertainty should explain why the plots are not regular and smooth. Fortunately for us, the uncertainty is not great enough to impact the conclusions made. JCuda is still faster on most cases than Aparapi, even taking into account the incertainty.

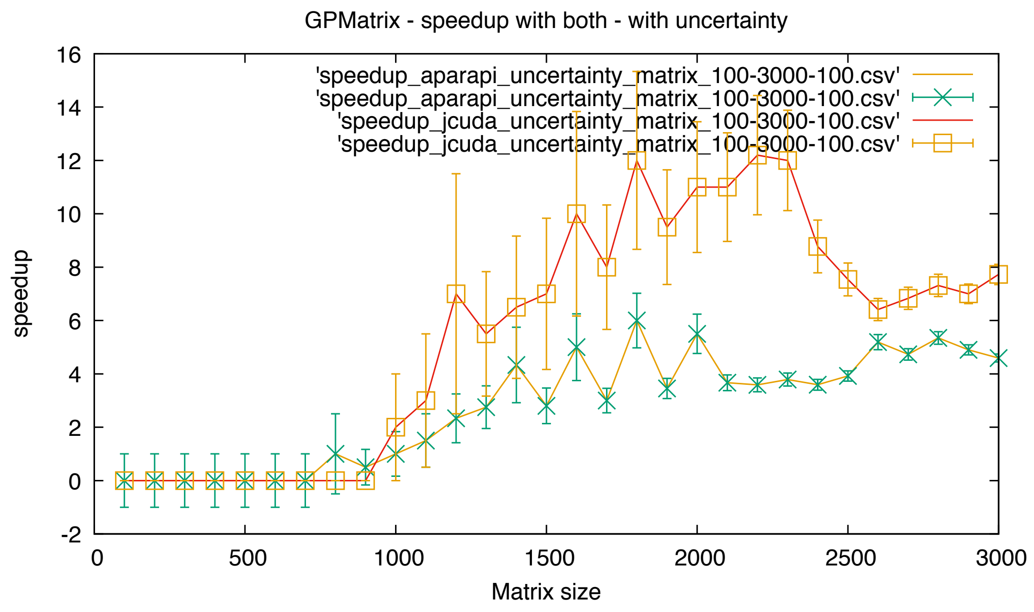


Figure E.1: Speedup of both Aparapi and JCuda on the matrix multiplication, showing uncertainty

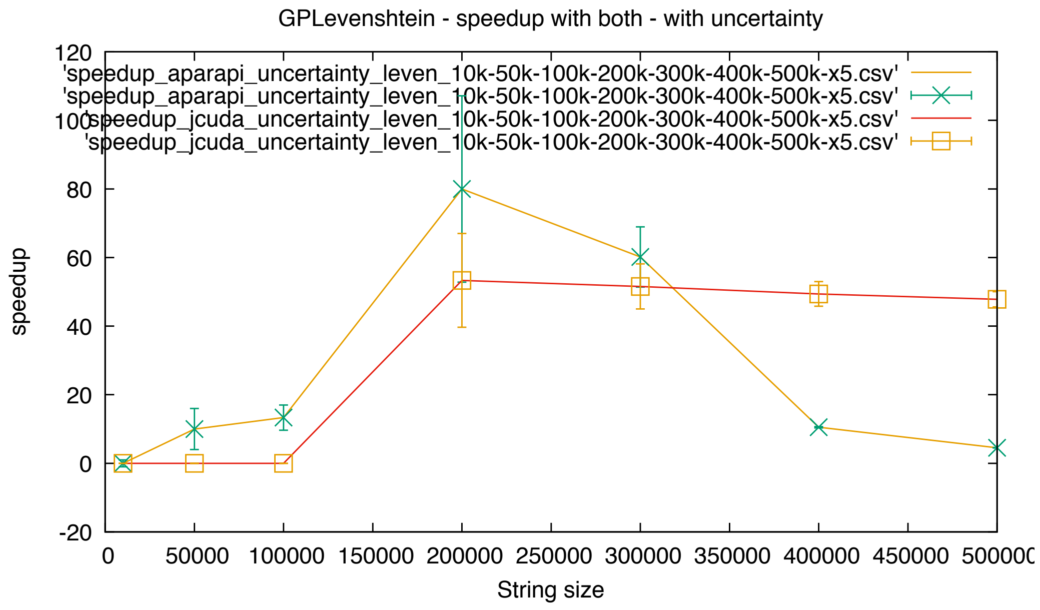


Figure E.2: Speedup of both Aparapi and JCuda on the Levenstein distance computation, showing uncertainty

Appendix F

Raw values

This appendix contains the raw values used for the plots in this document.

F.1 Quick mesasure - matrix

On a single execution. Double precision on an Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

Matrix size	**Execution time [s]**
200	0
300	0
400	0
500	0
600	0
700	0
800	1
900	1
1000	2
1100	3
1200	7
1300	11
1400	13
1500	14
1600	20
1700	24
1800	36
1900	38
2000	44
2100	55
2200	61
2300	72
2400	79
2500	98
2600	109
2700	123
2800	139
2900	147
3000	170

F.2 Quick measure - Leven

On a single execution. On an Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

String size	**Execution time [s]**
50000	10
100000	40
200000	160
300000	361
400000	642
500000	1004

F.3 Aparapi - matrix

Average of 5 executions. Double precision - Nvidia Tesla C2075.

Matrix size	**Execution time [s]**
200	1
300	1
400	1
500	1
600	1
700	1
800	1
900	2
1000	2
1100	2
1200	3
1300	4
1400	3
1500	5
1600	4
1700	8
1800	6
1900	11
2000	8
2100	15
2200	17
2300	19
2400	22
2500	25
2600	21
2700	26
2800	26
2900	30
3000	37

F.4 Aparapi - Leven

Average of 5 executions. Nvidia Tesla C2075.

String size	**Execution time [s]**
50000	1
100000	3
200000	2
300000	6
400000	61
500000	222

F.5 JCuda - matrix

Average of 5 executions. Double precision - Nvidia Tesla C2075.

Matrix size	**Execution time [s]**
200	0
300	0
400	0
500	0
600	0
700	0
800	0
900	0
1000	1
1100	1
1200	1
1300	2
1400	2
1500	2
1600	2
1700	3
1800	3
1900	4
2000	4
2100	5
2200	5
2300	6
2400	9
2500	13
2600	17
2700	18
2800	19
2900	21
3000	22

F.6 JCuda - Leven

Average of 5 executions. Nvidia Tesla C2075.

String size	**Execution time [s]**
50000	0
100000	0
200000	3
300000	7
400000	13
500000	21

F.7 Speedup Aparapi matrix

Average of 5 executions. Nvidia Tesla C2075 versus Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

Matrix size	**Speedup**
200	0
300	0
400	0
500	0
600	0
700	0
800	1.00000000000000000000
900	.50000000000000000000
1000	1.00000000000000000000
1100	1.50000000000000000000
1200	2.33333333333333333333
1300	2.75000000000000000000
1400	4.33333333333333333333
1500	2.80000000000000000000
1600	5.00000000000000000000
1700	3.00000000000000000000
1800	6.00000000000000000000
1900	3.45454545454545454545
2000	5.50000000000000000000
2100	3.66666666666666666666
2200	3.58823529411764705882
2300	3.78947368421052631578
2400	3.59090909090909090909
2500	3.92000000000000000000
2600	5.19047619047619047619
2700	4.73076923076923076923
2800	5.34615384615384615384
2900	4.90000000000000000000
3000	4.59459459459459459459

F.8 Speedup Aparapi Leven

Average of 5 executions. Nvidia Tesla C2075 versus Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

String size	**Speedup**
50000	10.00000000000000000000
100000	13.33333333333333333333
200000	80.00000000000000000000
300000	60.16666666666666666666
400000	10.52459016393442622950
500000	4.52252252252252252252

F.9 Speedup JCuda matrix

Average of 5 executions. Nvidia Tesla C2075 versus Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

Matrix size	**Speedup**
200	0
300	0
400	0
500	0
600	0
700	0
800	0
900	0
1000	2.00000000000000000000
1100	3.00000000000000000000
1200	7.00000000000000000000
1300	5.50000000000000000000
1400	6.50000000000000000000
1500	7.00000000000000000000
1600	10.00000000000000000000
1700	8.00000000000000000000
1800	12.00000000000000000000
1900	9.50000000000000000000
2000	11.00000000000000000000
2100	11.00000000000000000000
2200	12.20000000000000000000
2300	12.00000000000000000000
2400	8.77777777777777777777
2500	7.53846153846153846153
2600	6.41176470588235294117
2700	6.83333333333333333333
2800	7.31578947368421052631
2900	7.00000000000000000000
3000	7.72727272727272727272

F.10 Speedup JCuda Leven

Average of 5 executions. Nvidia Tesla C2075 versus Intel(R) Xeon(R) CPU E5-2609 v2 2.50GHz.

String size	**Speedup**
50000	0
100000	0
200000	53.33333333333333333333
300000	51.57142857142857142857
400000	49.38461538461538461538
500000	47.80952380952380952380

Appendix G

Vanilla matrix measure comparison

The following plot (figure G.1) is a comparison of the base values taken as a reference for the test battery concerning the matrix multiplication on a vanilla Java code.

On the graph is plotted the base values used to compute the speedup and a measure based on the average of 5 executions. This plot demonstrates that no edge cases happened while taking the base values. The two plots show a similar behavior and even sometimes the same values.

It is worth noting that the second measure took place 2 weeks after the first one.

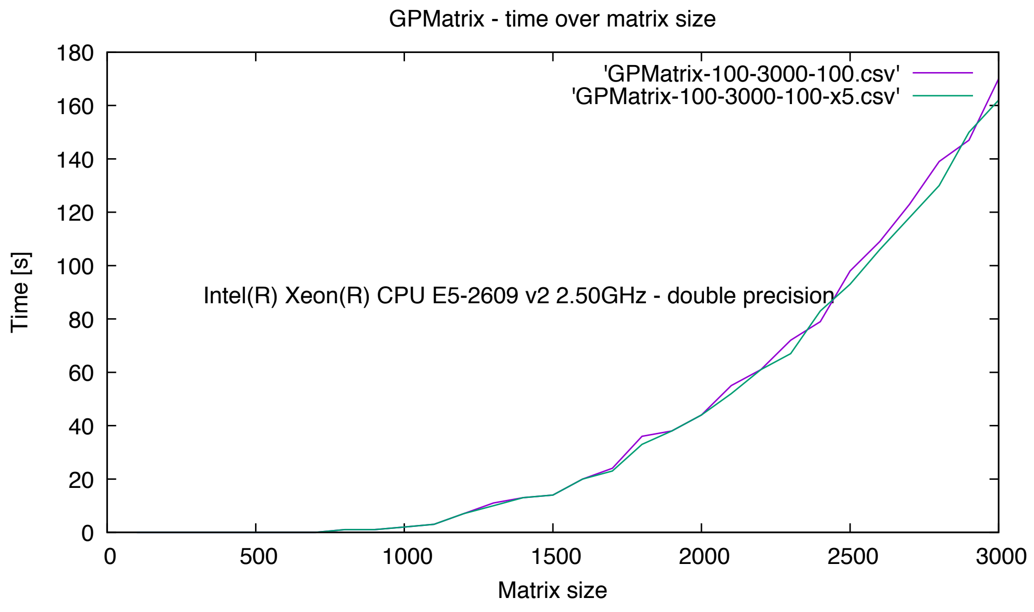


Figure G.1: Comparison of the base value with a measure performing an average on 5 executions.

Appendix H

External feedback survey for Aparapi

This appendix contains the feedback sheets used to evaluate Aparapi on the matrix multiplication and the Levenshtein distance computation respectively.

Le code ci-dessous calcul la distance de Levenstein entre deux chaînes de caractères - qui sont lues dans deux fichiers passés en paramètre. Une étape du calcul est faite sur le GPU via l'utilisation d'un framework. Après avoir pris connaissance du code, répondez aux 3 questions ci-dessous en cochant les cases.
***Note** : il n'est pas nécessaire de comprendre l'algorithme utilisé, mais plutôt l'utilisation du GPU.*

1. Quel est votre niveau d'expérience en programmation ?

Très grande expérience en programmation	1	2	3	4	5	Aucune expérience en programmation

2. Quel est notre niveau d'expérience en programmation GPU ?

Très grande expérience en programmation GPU	1	2	3	4	5	Aucune expérience en programmation GPU

3. Comment jugez-vous la difficulté à comprendre l'utilisation du GPU dans le code ?

Très difficile à comprendre	1	2	3	4	5	Très facile à comprendre

```java
import com.amd.aparapi.Kernel;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class GPLevenshtein {

  public static void main(String [] args) {

    // Params handling
    if(args.length < 2) {
      System.err.println("Needs two arguments: <file a> <file b>");
      System.exit(0);
    }
    String filea = args[0];
    String fileb = args[1];
    char[] a = null; // First string
    char[] b = null; // Second string

    // Reading files, filling a & b
    try {
      FileReader fra = new FileReader(filea);
      BufferedReader bra = new BufferedReader(fra);
      a = bra.readLine().toCharArray();
      FileReader frb = new FileReader(fileb);
      BufferedReader brb = new BufferedReader(frb);
      b = brb.readLine().toCharArray();
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }

    // --------------------------------
    long startTime = System.currentTimeMillis();
    // --- Start of benchmark zone --->
    new AparapiLevenshtein(a,b).execute(a.length+1);
    // <--- End of benchmark zone -----
    long stopTime = System.currentTimeMillis();
    // --------------------------------

    // Benchmark time elapsed computation
    long elapsedTime = stopTime - startTime;

    // Output
    System.out.println(elapsedTime/1000);
  }

}
```

```

// Is the class that sends job on the GPU
class AparapiLevenshtein extends Kernel {

  char[] a; // The first string
  char[] b; // The second string

  int[] costs; // Will contain the result
  int blen;    // https://github.com/steelted/aparapi/issues/117

  public AparapiLevenshtein(char[] a, char[] b) {

    this.a = a;
    this.b = b;
    this.blen = b.length;

    costs = new int [b.length + 1];
    for (int j = 0; j < costs.length; j++)
      costs[j] = j;

  }

  @Override
  // Kernel method
  public void run() {
    int i = getGlobalId(); // The id of the thread
    if(i==0) return;

    costs[0] = i;
    int nw = i - 1;
    for (int j = 1; j <= blen; j++) {
      int cj = min(1 + min(costs[j], costs[j - 1]), a[i - 1] == b[j - 1] ? nw : nw + 1);
      nw = costs[j];
      costs[j] = cj;
    }

  }

  public int min(int a, int b) {
    return a < b ? a : b;
  }

}
```

Le code ci-dessous calcul la multiplication entre deux matrices carrées – dont la taille est passée en paramètre. Le calcul s’effectue sur le GPU à l’aide du framework Aparapi. Après avoir pris connaissance du code, répondez aux 3 questions ci-dessous en cochant les cases.
***Note** : il n’est pas nécessaire de comprendre l’algorithme utilisé, mais plutôt l’utilisation du GPU.*

1. Quel est votre niveau d’expérience en programmation ?

Très grande expérience en programmation	1	2	3	4	5	Aucune expérience en programmation

2. Quel est notre niveau d’expérience en programmation GPU ?

Très grande expérience en programmation GPU	1	2	3	4	5	Aucune expérience en programmation GPU

3. Comment jugez-vous la difficulté à comprendre l’utilisation du GPU dans le code ?

Très difficile à comprendre	1	2	3	4	5	Très facile à comprendre

```java
import java.util.Random;
import com.amd.aparapi.Kernel;

public class GPMatrix {

  public static int verbose = 0; // level 0,1,2
  public static int size;

  public static void main(String[] args) {

    // Params handling
    if(args.length < 1) {
      System.err.println("Needs size argument");
      System.exit(0);
    }
    size = Integer.valueOf(args[0]);

    // Matrix declaration
    double[][] A = new double[size][size];
    double[][] B = new double[size][size];
    double[][] C = new double[size][size];

    matrixInit(A,B);

    // --------------------------------
    long startTime = System.currentTimeMillis();
    // --- Start of benchmark zone --->
    new AparapiMatrixMul(A,B,C, size).execute(size);
    // <--- End of benchmark zone -----
    long stopTime = System.currentTimeMillis();
    // --------------------------------

    // Benchmark time elapsed computation
    long elapsedTime = stopTime - startTime;

    // Output
    if(verbose > 0) System.out.println("Time elapsed: " + elapsedTime/1000 + "s");
    System.out.println(elapsedTime/1000);

  }

  // Matrix random initialization of A and B
  public static void matrixInit(double[][] A, double[][] B) {
    Random r = new Random();
    for(int row=0; row < size; row++) {
      for(int col=0; col < size; col++) {
        A[row][col] = r.nextDouble();
        B[row][col] = r.nextDouble();
      }
    }
  }

}

// Is the class that sends job on the GPU
class AparapiMatrixMul extends Kernel {

  double[][] A; // Matrix A
  double[][] B; // Matrix B
  double[][] C; // Matrix C, the result of AxB
  int size;

  public AparapiMatrixMul(double[][] A, double[][] B, double[][] C, int size) {
    this.A = A; this.B = B; this.C = C;
    this.size = size;
  }

  @Override
  // Kernel method
  public void run() {
    int i = getGlobalId(); // The id of the thread

    for(int j=0; j < size; j++) {
      double sum = 0.0;
      for(int k=0; k < size; k++) {
        sum += A[i][k] * B[k][j];
      }
      C[i][j] = sum;
    }
  }

}
```

Appendix I

External feedback survey for JCuda

This appendix contains the feedback sheets used to evaluate JCuda on the matrix multiplication and the Levenshtein distance computation respectively.

Le code ci-dessous calcul la multiplication entre deux matrices carrées – dont la taille est passée en paramètre. Le calcul s'effectue sur le GPU à l'aide du framework JCuda. Le kernel (méthode s'effectuant sur le CPU) est dans un fichier à part.
Après avoir pris connaissance du code, répondez aux 3 questions ci-dessous en cochant les cases.
***Note** : il n'est pas nécessaire de comprendre l'algorithme utilisé, mais plutôt l'utilisation du GPU.*

1. Quel est votre niveau d'expérience en programmation ?

Très grande expérience en programmation	1	2	3	4	5	Aucune expérience en programmation

2. Quel est notre niveau d'expérience en programmation GPU ?

Très grande expérience en programmation GPU	1	2	3	4	5	Aucune expérience en programmation GPU

3. Comment jugez-vous la difficulté à comprendre l'utilisation du GPU dans le code ?

Très difficile à comprendre	1	2	3	4	5	Très facile à comprendre

```java
import java.util.Random;
import jcuda.*;
import jcuda.runtime.*;
import jcuda.driver.*;
import static jcuda.driver.JCudaDriver.*;

public class GPMatrix {

  public static int verbose = 0; // level 0,1,2
  public static int size;

  public static void main(String[] args) {

    // Params handling
    if(args.length < 1) {
      System.err.println("Needs size argument");
      System.exit(0);
    }
    size = Integer.valueOf(args[0]);

    // Matrix declaration
    double[] A = new double[size * size];
    double[] B = new double[size * size];
    double[] C = new double[size * size];

    matrixInit(A,B);


    // Enable exception
    JCudaDriver.setExceptionsEnabled(true);

    // Initialize the driver and create a context for the first device
    cuInit(0);
    CUdevice device = new CUdevice();
    cuDeviceGet(device, 0);
    CUcontext context = new CUcontext();
    cuCtxCreate(context, 0, device);

    // Load the CUDA kernel ptx file
    CUmodule module = new CUmodule();
    cuModuleLoad(module, "../JCudaMatrix.ptx");

    // Obtain a function pointer to the kernel function
    CUfunction function = new CUfunction();
    cuModuleGetFunction(function, module, "mul");

    // Allocate the device input data, and copy the host input data
    // to the device
    int ptrSize = size * size * Sizeof.DOUBLE;
    CUdeviceptr deviceInputA = new CUdeviceptr();
    cuMemAlloc(deviceInputA, ptrSize);
    cuMemcpyHtoD(deviceInputA, Pointer.to(A), ptrSize);

    CUdeviceptr deviceInputB = new CUdeviceptr();
    cuMemAlloc(deviceInputB, ptrSize);
    cuMemcpyHtoD(deviceInputB, Pointer.to(B), ptrSize);

    CUdeviceptr deviceOutput = new CUdeviceptr();
    cuMemAlloc(deviceOutput, ptrSize);

    // Set up the kernel parameters: A pointer to an array
    // of pointers which points to the actual values
    Pointer kernelParameters = Pointer.to(
      Pointer.to(deviceInputA),
      Pointer.to(deviceInputB),
      Pointer.to(deviceOutput),
      Pointer.to(new int[]{size})
    );

    // Kernel call parameters
    int blockSizeX = 256;
    int gridSizeX = (int)Math.ceil((double)size / blockSizeX);

    // --------------------------------
    long startTime = System.currentTimeMillis();
    // --- Start of benchmark zone --->
    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
    );
    cuCtxSynchronize();
    // <--- End of benchmark zone -----
    long stopTime = System.currentTimeMillis();
    // --------------------------------

    // Benchmark time elapsed computation
    long elapsedTime = stopTime - startTime;

    // Output
    if(verbose > 0) System.out.println("Time elapsed: " + elapsedTime/1000 + "s");
    System.out.println(elapsedTime/1000);

  }

  // Matrix random initialization of A and B
  public static void matrixInit(double[] A, double[] B) {
    Random r = new Random();
    for(int row=0; row < size; row++) {
      for(int col=0; col < size; col++) {
        A[ (row*size)+col ] = r.nextDouble();
        B[ (row*size)+col ] = r.nextDouble();
      }
    }
  }


}

```

Fichier contenant le kernel (méthode s'exécutant sur le GPU) :

```
extern "C"
__global__ void mul(double* A, double* B, double* C, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < size) {
    // compute a column
    for(int j=0; j < size; j++) {
      double sum = 0.0;
      for(int k=0; k < size; k++) {
        sum += A[ (i*size)+k ] * B[ (k*size)+j ];
      }
      C[ (i*size)+j ] = sum;
    }
    // end of column computing
  }
}
```

Le code ci-dessous calcul la distance de Levenstein entre deux chaînes de caractères - qui sont lues dans deux fichiers passés en paramètre. Une étape du calcul est faite sur le GPU via l'utilisation du framework JCuda. Le kernel (méthode s'effectuant sur le GPU) est dans un fichier à part.
Après avoir pris connaissance du code, répondez aux 3 questions ci-dessous en cochant les cases.
***Note** : il n'est pas nécessaire de comprendre l'algorithme utilisé, mais plutôt l'utilisation du GPU.*

1. Quel est votre niveau d'expérience en programmation ?

Très grande expérience en programmation	1	2	3	4	5	Aucune expérience en programmation

2. Quel est notre niveau d'expérience en programmation GPU ?

Très grande expérience en programmation GPU	1	2	3	4	5	Aucune expérience en programmation GPU

3. Comment jugez-vous la difficulté à comprendre l'utilisation du GPU dans le code ?

Très difficile à comprendre	1	2	3	4	5	Très facile à comprendre

```java
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import jcuda.*;
import jcuda.runtime.*;
import jcuda.driver.*;
import static jcuda.driver.JCudaDriver.*;

public class GPLevenshtein {

  public static void main(String [] args) {

    if(args.length < 2) {
      System.err.println("Needs two arguments: <file a> <file b>");
      System.exit(0);
    }
    String filea = args[0];
    String fileb = args[1];
    char[] a = null;
    char[] b = null;

    try {
      FileReader fra = new FileReader(filea);
      BufferedReader bra = new BufferedReader(fra);
      a = bra.readLine().toCharArray();
      FileReader frb = new FileReader(fileb);
      BufferedReader brb = new BufferedReader(frb);
      b = brb.readLine().toCharArray();
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }

    // Initialize the costs array
    int[] costs = new int[a.length+1];

    // Enable exception
    JCudaDriver.setExceptionsEnabled(true);

    // Initialize the driver and create a context for the first device
    cuInit(0);
    CUdevice device = new CUdevice();
    cuDeviceGet(device, 0);
    CUcontext context = new CUcontext();
    cuCtxCreate(context, 0, device);

    // Load the CUDA kernel ptx file
    CUmodule module = new CUmodule();
    cuModuleLoad(module, "../JCudaLevenstein.ptx");

    // Obtain a function pointer to the kernel function
    CUfunction function = new CUfunction();
    cuModuleGetFunction(function, module, "leven");

    // Allocate the device input data, and copy the host input data
    // to the device
    int ptrSize = a.length * Sizeof.CHAR;
    CUdeviceptr deviceInputA = new CUdeviceptr();
    cuMemAlloc(deviceInputA, ptrSize);
    cuMemcpyHtoD(deviceInputA, Pointer.to(a), ptrSize);

    CUdeviceptr deviceInputB = new CUdeviceptr();
    cuMemAlloc(deviceInputB, ptrSize);
    cuMemcpyHtoD(deviceInputB, Pointer.to(b), ptrSize);

    CUdeviceptr deviceInputCosts = new CUdeviceptr();
    cuMemAlloc(deviceInputCosts, a.length * Sizeof.INT + 1);
    cuMemcpyHtoD(deviceInputCosts, Pointer.to(costs), a.length * Sizeof.INT + 1);

    // Set up the kernel parameters: A pointer to an array
    // of pointers which points to the actual values
    Pointer kernelParameters = Pointer.to(
      Pointer.to(deviceInputA),
      Pointer.to(deviceInputB),
      Pointer.to(deviceInputCosts),
      Pointer.to(new int[]{a.length})
    );

    // Kernel call parameters
    int blockSizeX = 256;
    int gridSizeX = (int)Math.ceil((double)a.length / blockSizeX);

    // --------------------------------
    long startTime = System.currentTimeMillis();
    // --- Start of benchmark zone --->
    for(int i = 0; i < costs.length; i++) costs[i] = i;
    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
    );
    cuCtxSynchronize();
    // <--- End of benchmark zone -----
    long stopTime = System.currentTimeMillis();
    // --------------------------------

    // Benchmark time elapsed computation
    long elapsedTime = stopTime - startTime;

    // Output
    System.out.println(elapsedTime/1000);
  }
```

Fichier contenant le kernel (méthode s'effectuant sur le GPU).

```
extern "C"

__global__ void leven(char* a, char* b, char* costs, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i > 0 && i < size) {

    costs[0] = i;
    int nw = i - 1;
    for(int j = 1; j <= size; j++) {
      int firstMin = costs[j] < costs[j-1] ? costs[j] : costs[j-1];
      // This line is hard to read due to the lack of min() function
      int secondMin = 1 + firstMin < a[i - 1] == b[j - 1] ?
        nw : nw + 1 ? 1 + firstMin : a[i - 1] == b[j - 1] ? nw : nw + 1;
      int cj = secondMin;
      nw = costs[j];
      costs[j] = cj;
    }
  }

}
```