Evaluation of Java for General Purpose GPU Computing

Jorge Docampo, Sabela Ramos, Guillermo L. Taboada, Roberto R. Expósito, Juan Touriño, Ramón Doallo
Computer Architecture Group
Department of Electronics and Systems
University of A Coruña (Spain)
Email: {jorge.docampo, sramos, taboada, rreye, juan, doallo}@udc.es

***Abstract*—The presence of many-core units as accelerators has been increasing due to their ability to improve the performance of highly parallel workloads. General Purpose GPU (GPGPU) computing has allowed the graphical units to emerge as successful co-processors that can be employed to improve the performance of many different non-graphical applications with high parallel requirements, which make them suitable for many High Performance Computing workloads. While the main libraries developed to exploit the massive parallel capacity of GPUs are oriented to C/C++ programmers, there have been several efforts to extend this support to other languages. Among them, Java stands out for being one of the most extended languages and there are multiple projects that try to enable Java to take advantage of GPGPU computing. In this scenario, this paper presents an evaluation of the most relevant among the current solutions that exploit GPGPU computing in Java.**

***Keywords*-Java; General Purpose GPU computing (GPGPU); jCuda; Aparapi; High Performance Computing;**

I. Introduction

The interest in exploiting Graphical Processing Units (GPUs) for non-graphical purposes has been motivated by their massively parallel architecture, their floating point capacity and their high memory bandwidth. Thus, as GPUs are suitable to manage parallel workloads that are typical in scientific applications, the High Performance Computing (HPC) community has adopted GPUs as many-core accelerators, and the General Purpose GPU (GPGPU) computing has become increasingly popular. The TOP 500 list of the most powerful supercomputers [1] reflects this trend and, from the 12.4% of supercomputers that use many-core accelerators (which were only 7.8% one year ago), 85% use GPUs.

The main programming models for GPGPU computing were developed as libraries to be used from C/C++. However, since Java is one of the leading languages both in industry and academia, several projects have emerged to make Java able to benefit from offloading workloads on the GPU. Java is also an increasing alternative for HPC programming [2] since its performance has improved in the last years, narrowing the gap with native compiled languages like C or Fortran, and it has several interesting characteristics like its ease of use, object orientation, multi-threading and built-in network support, automatic memory management, platform independence, portability and a wide community of developers.

This paper aims to provide a reference evaluation of the projects that enable GPGPU computing in Java, with a compilation of the most relevant and active solutions among the existing ones. Additionally, this work includes a comparative evaluation and analysis of their features and performance. To our knowledge this is the first paper that evaluates current Java GPGPU solutions and compare their performance against native and Java codes.

The rest of the paper organizes as follows. Section II provides an analysis of the most relevant projects for GPGPU computing in Java, Section III includes a comparative evaluation of the performance of the presented solutions and Section IV presents the main concluding remarks.

II. Current Solutions for General Purpose GPU Computing in Java

The massively parallel architecture of GPUs, together with their floating point capacity, has motivated the growth of General-Purpose computing on GPUs (GPGPU) [3], along with different programming models, like Compute Unified Device Architecture (CUDA) [4] or Open Computing Language (OpenCL) [5], to enable the use of GPUs as many-core accelerators in non-graphical workloads. Thus, it has raised the adoption of GPUs as accelerators in HPC [6] environments, since many scientific applications present a huge degree of parallelism that can take advantage of GPU's features.

These GPGPU models are intended to be used as extensions to C/C++ codes, whereas other languages like Java must resort to wrappers (via JNI, Java Native Interface) to be able to exploit GPUs as accelerators. Nevertheless, the interest in Java for HPC has been growing due to its increasing performance and its appealing features like multithreading or network support [2]. Thus, several projects have appeared to support the use of Java in GPGPU programming.

Among them, we can distinguish two approaches, the ones that provide Java bindings to a lower level language (CUDA or OpenCL) or those with a user-friendly API that abstracts GPU programming along with a runtime system which translates Java bytecode into CUDA or OpenCL in a transparent manner. While Java bindings are meant to provide better performance, increasing programmability makes it possible to find a tradeoff between performance and productivity.

Table I summarizes the most relevant projects for GPGPU computing in Java also indicating which is the underlying native library. Regarding CUDA-related projects, JCUDA [7] has its own interface to invoke certain CUDA functions and user developed kernels. Nevertheless, it is not included in the User-friendly group since it still requires low-level programming skills and certain knowledge of CUDA functions.

jCuda [8] is the most active Java GPGPU project. It provides a direct wrapper over CUDA 4.2 runtime and driver API, allowing the direct interaction with the device, including memory management and allowing the launch of CUDA kernels from Java. The main strength of this project is that it provides support for several optimized libraries from CUDA like CUBLAS (CUDA Basic Linear Algebra Subprograms), CUFFT (CUDA Fast Fourier Transforms), CUDPP (CUDA Data Parallel Primitives), CURAND (CUDA Random Number Generation), CUSPARSE (CUDA Sparse Matrix) and NPP (NVIDIA Performance Primitives). The jCuda API consists of a group of static methods which are very similar to the native library functions since the aim of jCuda is to keep the API as close to the original as possible, including also functions in order to use user defined kernels in CUDA language, as well as pointer handling functions.

Java-GPU [9] introduces directives to offload Java code into the GPU, whereas Rootbeer [10], which has been published recently, provides a specific high-level API for Java and translates the generated bytecode into CUDA.

Table I
AVAILABLE SOLUTIONS FOR GPGPU COMPUTING IN JAVA

	Java bindings	**User-friendly**
CUDA	JCUDA [7] jCuda [8]	Java-GPU [9] Rootbeer [10]
OpenCL	JOCL [11] JogAmp's JOCL [13]	Aparapi [12]

Java OpenCL binding solutions include JOCL [11] and JogAmp's JOCL [13]. The main difference between them is that while the former provides support for OpenCL 1.2, the latter only handles OpenCL 1.1.

Finally, Aparapi [12] is the most up-to-date Java OpenCL project which provides OpenCL 2.1 support. The Aparapi programmer is provided with a high-level API to express data parallel workloads in Java, being released from all the GPU implementation details. Nevertheless, the programmer should have some notions of how the GPU works in order to perform an efficient distribution of the work and obtain a significant performance, but no knowledge at all of the OpenCL language is needed. The runtime system will translate these data parallel workloads to OpenCL and will offload them on a GPU or a pool of threads. Aparapi is supported by AMD and its source code has been released with a GPL license.

III. PERFORMANCE EVALUATION

This section assesses the performance of two representative Java GPGPU projects, jCuda and Aparapi, selected for being the most active and for their representativeness, jCuda among the direct wrapper implementations over native libraries and Aparapi among the user-friendly-approaches. Moreover, jCuda is based on CUDA whereas Aparapi relies on OpenCL, but this fact is not especially relevant. A previous work [14] on the evaluation of CUDA and OpenCL, on the experimental testbed used for the performance evaluation presented in this section, has shown that CUDA and OpenCL are able to provide roughly the same performance, therefore the actual implementation of a given code is the main reason of performance differences among CUDA and OpenCL.

A. Experimental Configuration

The testbed used for the performance evaluation is an x86_64 server with the following characteristics:

Table II
DESCRIPTION OF THE GPU-BASED TESTBED

CPU	1 × Intel(R) Xeon hexacore X5650 @ 2.67GHz
CPU performance	64.08 GFLOPS DP (10.68 GFLOPS DP per core)
GPU	1 × NVIDIA Tesla "Fermi" M2050
GPU performance	515 GFLOPS DP
Memory	12 GB DDR3 (1333 MHz)
OS	Debian GNU/Linux, kernel 3.2.0-3
CUDA	version 4.2 SDK Toolkit
JDK version	OpenJDK 1.6.0_24

The evaluation of Java GPGPU has been done using representative GPGPU synthetic kernels, that are code snippets which provide with widely extended basic building blocks in HPC applications (e.g., a matrix multiplication kernel). The synthetic kernels used for the evaluation of Java GPGPU have been selected from the benchmark suites Scalable HeterOgeneus Computing (SHOC) [15] and the Java-GPU distribution examples. On the one hand, the SHOC suite determines the computational performance of the system with the aid of application kernels. Table III presents the four synthetic kernels selected. We have developed the Java, jCuda and Aparapi implementations, which rely on the CPU, and the CUDA and OpenCL on the GPU, respectively, allowing the comparative analysis of their performance.

The reported performance results of the level 1 kernels (see Figures 1 - 3) are the average of 5 runs, observing reduced variability among measurements.

Table III
SELECTED SYNTHETIC KERNELS

Kernel	**Suite**	**Measure Unit**	**Description**
MaxFlops	SHOC lvl.0	GFLOPS	Peak GFlops
GEMM	SHOC lvl.1	GFLOPS	Matrix multiplication
Stencil2D	SHOC lvl.1	Time & GFLOPS	A two-dimensional nine point stencil calculation
FFT	SHOC lvl.1	Time & GFLOPS	Fast Fourier Transform

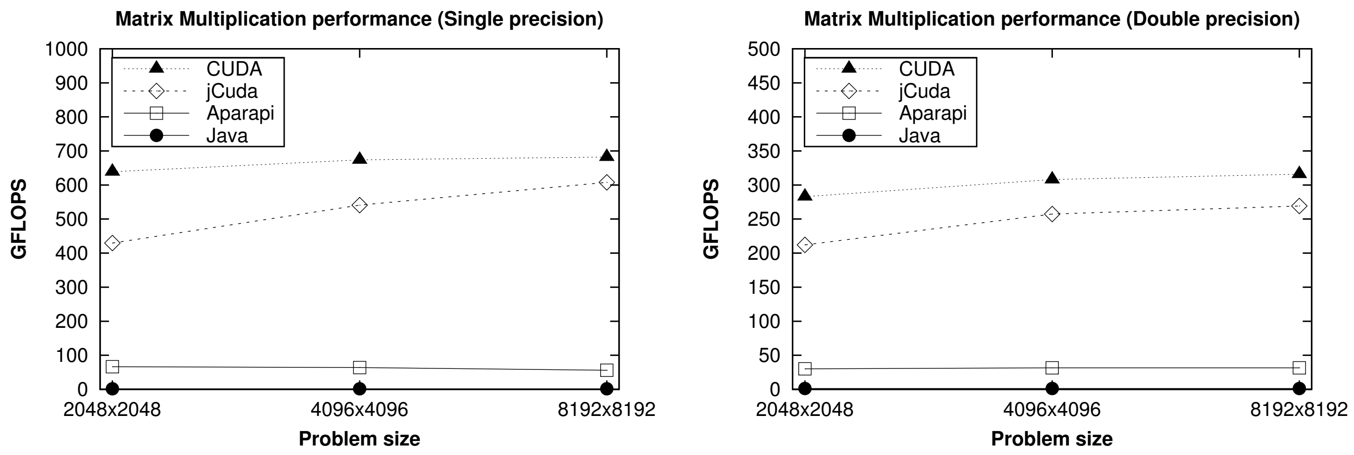

Figure 1. Matrix multiplication kernel performance

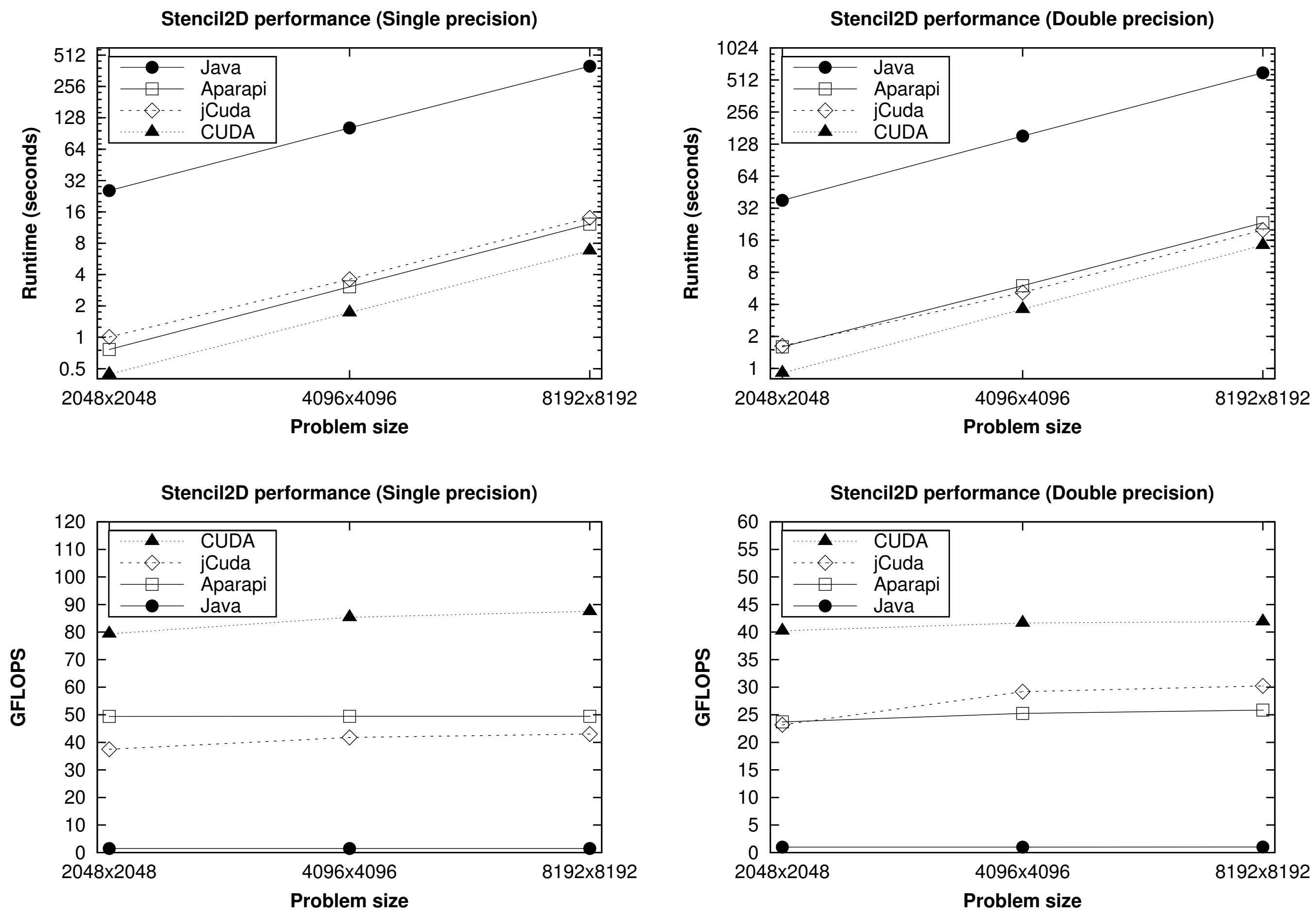

Figure 2. Stencil 2D kernel performance

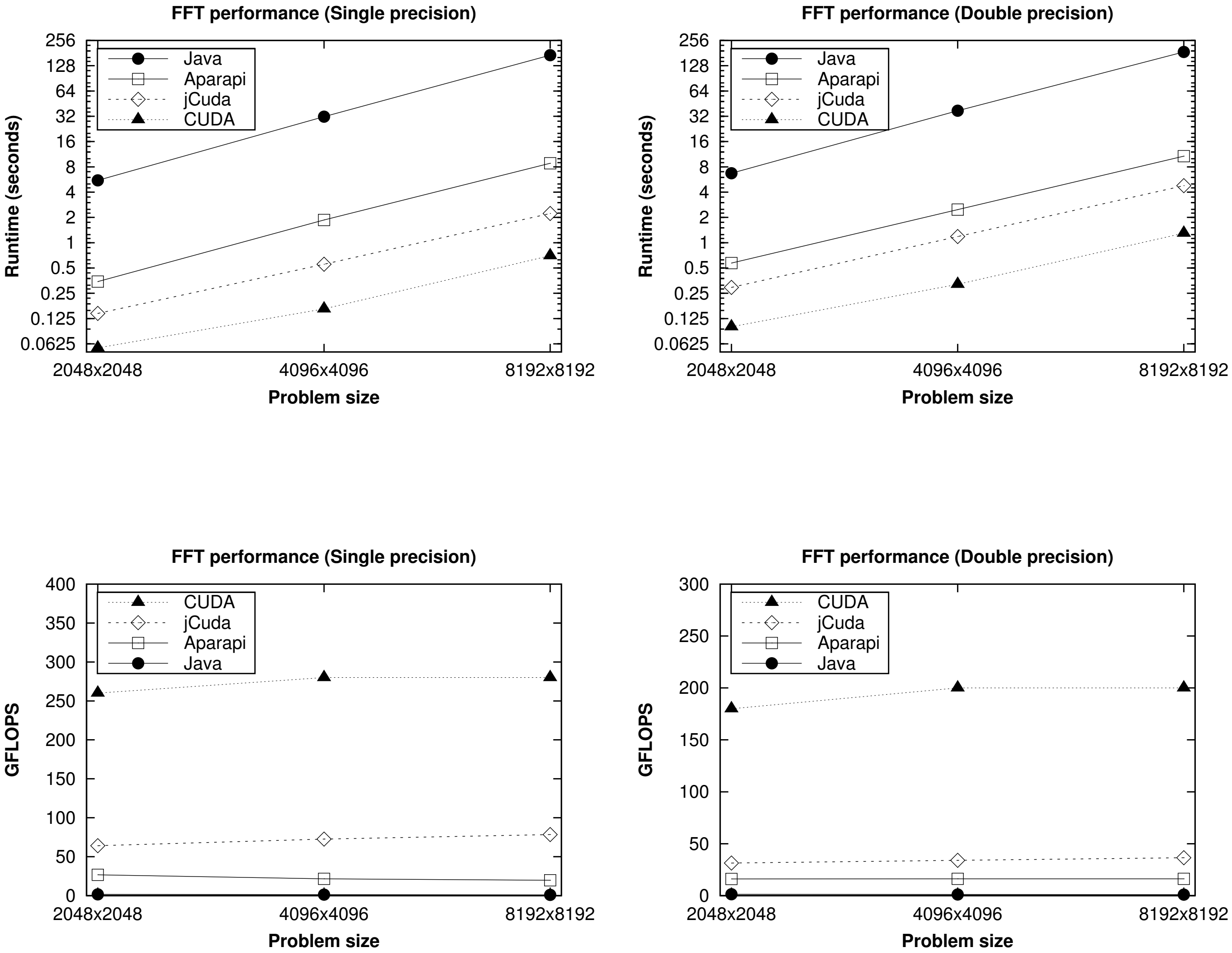

Figure 3. FFT kernel performance

B. Analysis of Experimental Results

Table IV shows an analysis of the raw peak performance differences in GFLOPS for Aparapi and jCuda compared to the SHOC results. The benchmark used shows that the difference between the peak flops obtained on both Aparapi and jCuda in the case of single precision is not a handicap at all, being between 97-89% of the result obtained in SHOC. But for double precision these rates fall down to 76-66%, but still acceptable for both cases.

Table IV
MAXFLOPS PERFORMANCE

	GFLOPS			
	Simple precision		**Double precision**	
SHOC	1008.16	100%	509.19	100%
jCuda	982.68	97.47%	387.10	76.02%
Aparapi	902.51	89.52%	338.72	66.52%

Figure 1 presents the performance results, in terms of GFLOPS, of the CUDA, jCuda, Aparapi and Java implementations of the matrix multiplication kernel. Here CUDA achieves the highest performance, followed by jCUDA. As both implementations rely on the same matrix multiplication routine of CUBLAS the performance gap between them is due to the overhead of the data movements between Java and the GPU. In fact, jCuda suffers a higher penalty for 2048x2048 problem size due to the high start-up overhead of the Java-GPU data copy, but as the problem size increases jCuda is able to obtain around 85-90% of the CUDA performance.

As the standard JVMs do not support GPUs directly Java performance results have been obtained on the CPU, running a pure Java (without relying on native methods) matrix multiplication code which yields around 1 GFLOPS both for single and double precision. Although Java would be able to achieve higher performance calling any BLAS library with Java bindings (e.g., Intel Math Kernel Library, MKL), we opt for using as baseline a standard and fully portable Java code.

Finally, Aparapi was introduced in the standard Java kernel implementation and the observed performance benefits are highly important, up to 40 times speedup for single precision executions (from 1.6 GFLOPS for Java up to 64 GFLOPS for Aparapi) and up to 23 times speedup for double precision operations (from 1.3 GFLOPS for Java up to 31 GFLOPS for Aparapi). In case Aparapi does not find a GPU in the system it would run the code using the CPU, so portability is not compromised with this solution.

Figure 2 presents the performance results, in terms of Time (seconds) and GFLOPS, of the CUDA, jCuda, Aparapi and Java implementations of the Stencil2D kernel. Once again CUDA achieves the highest performance and Java on the CPU is around 40-60 times slower. However, jCuda and Aparapi are able to achieve around 50-75% of the native CUDA performance, accelerating up to 33 times the performance of Java.

Here jCuda does not rely on a CUDA library, so its performance is not as close to CUDA as for the matrix multiplication kernel. Moreover, as both jCuda and Aparapi implement the same algorithm both achieve similar performance results. However, the productivity of Aparapi is much higher as it has been much easier to develop the Aparapi code than the jCuda implementation. This lower time to solution and the significant speedup achieved suggest that Aparapi is the best Java GPGPU option when jCuda can not rely on an optimized CUDA library such as CUBLAS or CUFFT.

Figure 3 presents the performance results, in terms of Time (seconds) and GFLOPS, of the CUDA, jCuda, Aparapi and Java implementations of the FFT kernel. In this case jCuda relies also on CUFFT but the kernel is not as computationally intensive as the matrix multiplication (this FFT algorithm has a complexity of $nlog(n)$ whereas matrix multiplication has a complexity of $O(n^3)$). Thus, Java-GPU data movements have higher impact on the overall performance of jCuda which only achieves around 25-30% of CUDA raw performance.

Aparapi FFT is around 20 times faster than Java on the CPU but at the same time is around 2-3 times slower than jCuda, and more than an order of magnitude slower than CUDA. Nevertheless, for Java programmers Aparapi represents a portable and productive option for accelerating their standard Java applications in presence of GPUs. However, when performance is critical jCuda and even writing CUDA native methods, accessible through JNI, is the way to go, although at the cost of losing portability and a higher time-to-solution.

C. Analysis of Code

The analysis of productivity has been done in terms of expertise needed to develop a simple application for the three options.

On the one hand, Figures 4 and 5 show the original Java code for vector addition and its translation to Aparapi. In this case it is only needed to create a kernel object, override the run function so that each thread calculates a single value of the solution vector, and set the number of threads that will execute the function.

```
final float inA[] = .... // get a float array of data from somewhere
final float inB[] = .... // get a float array of data from somewhere
final float result = new float[inA.length];
for (int i=0; i<array.length; i++){
    result[i]=intA[i]+inB[i];
}
```

Figure 4. Java code for the `Vector addition`

```
Kernel kernel = new Kernel(){
   @Override public void run() {
      int i= getGlobalId();
      result[i]=intA[i]+inB[i]; }
};
Range range = Range.create(result.length);
kernel.execute(range);
```

Figure 5. Aparapi code for the `Vector addition`

On the other hand, jCuda and CUDA would need a CUDA kernel with the same algorithm that the run function in the Aparapi code. However, the global (host) code will involve device memory management in both cases, which severely complicates the code development. Moreover, jCuda it also involves an initialization of both device and context, as well a compilation of the kernel code to a ptx file and its load.

```
__global__ void vecAdd(float *a, float *b,
                       float *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Figure 6. Kernel code for the `Vector addition` in jCuda and CUDA

```
int vectorAdd(int n, float *h_a, float *h_b, float *h_c)
{
    // Device input vectors
    float *d_a;
    float *d_b;
    //Device output vector
    float *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(float);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    // Copy host vectors to device
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);
    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    // Copy array back to host
    cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);
    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Figure 7. CUDA global C code for the `Vector addition`

```
void vectorAdd(int n, float[] h_a, float[] h_b,
                                   float[] h_c)
{
  JCudaDriver.setExceptionsEnabled(true);

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

  // 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 ptx file.
  CUmodule module = new CUmodule();
  cuModuleLoad(module, ptxFileName);

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

  // Allocate the device input data, and copy the
  // host input data to the device
  CUdeviceptr deviceInputA = new CUdeviceptr();
  cuMemAlloc(deviceInputA, n * Sizeof.FLOAT);
  cuMemcpyHtoD(deviceInputA, Pointer.to(h_a),
           n * Sizeof.FLOAT);
  CUdeviceptr deviceInputB = new CUdeviceptr();
  cuMemAlloc(deviceInputB, n * Sizeof.FLOAT);
  cuMemcpyHtoD(deviceInputB, Pointer.to(h_b),
           n * Sizeof.FLOAT);

  // Allocate device output memory
  CUdeviceptr deviceOutput = new CUdeviceptr();
  cuMemAlloc(deviceOutput, n * Sizeof.FLOAT);

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

  // Call the kernel function.
  int blockSizeX = 256;
  int gridSizeX = (int)Math.ceil((double)n / blockSizeX);
  cuLaunchKernel(function,
                gridSizeX,  1, 1,  // Grid dimension
                blockSizeX, 1, 1,  // Block dimension
                0, null,           // Shared mem size and stream
                kernelParameters,  // Kernel parameters
                null               // Extra parameters
               );
  cuCtxSynchronize();

  // Copy the device output to the host.
  cuMemcpyDtoH(Pointer.to(h_c), deviceOutput,
           n * Sizeof.FLOAT);

  // Clean up.
  cuMemFree(deviceInputA);
  cuMemFree(deviceInputB);
  cuMemFree(deviceOutput);
}
```

Figure 8. jCuda code for the `Vector addition`

IV. Conclusions

This paper has presented and analyzed the most relevant libraries that enables the use of Java in GPGPU computing. After selecting the most representative ones, jCuda and Aparapi, the performance evaluation has shown that, while Aparapi represents a good tradeoff between productivity and performance, jCuda requires more programming effort but, in exchange, it provides better performance results, especially when relying on optimized CUDA libraries such as CUBLAS and CUFFT, being able to rival with CUDA performance. The conclusions derived from this work are key to provide Java developers with an up-to-date information on the current solutions for GPGPU programming and their performance, which would increase definitively their productivity.

Acknowledgment

This work was supported by the Ministry of Science and Innovation of Spain [Project TIN2010-16735 and an FPU Grant AP2010-4348], and by the Galician Government [Program for the Consolidation of Competitive Research Groups, ref. 2010/6, and grant CN2012/211], partially supported by ERDF funds.

References

[1] Top 500 Supercomputers List, http://top500.org, [Last visited December 2012].

[2] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, “Java in the High Performance Computing Arena: Research, Practice and Experience,” *Science of Computing Programming*, vol. (in press), 2012.

[3] A. Leist, D. Playne, and K. Hawick, “Exploiting Graphical Processing Units for Data-Parallel Scientific Applications,” *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, 2009.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[5] J. Stone, D. Gohara, and S. Guochun, “OpenCL: a Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU Cluster for High Performance Computing,” in *Proc. 16th ACM/IEEE Conf. on Supercomputing (SC’04)*, Pittsburgh, PA, USA, 2004, p. 47 (12 pages).

[7] Y. Yan, M. Grossman, and V. Sarkar, “JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA,” in *Proc. 15th Intl. Euro-Par Conf. on Parallel Processing (Euro-Par’09)*, Delft, The Netherlands, 2009, pp. 887–899.

[8] jCuda. Java bindings for CUDA, http://jcuda.org, [Last visited December 2012].

[9] P. Calvert, “Parallelisation of Java for Graphics Processors,” Part II Dissertation, Computer Science Tripos, University of Cambridge, 2010.

[10] P. Pratt-Szeliga, J. Fawcett, and R. Welch, “Rootbeer: Seamlessly using GPUs from Java,” in *Proc. 14th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC’12)*, Liverpool, UK, 2012 (In press).

[11] jCuda. Java bindings for OpenCL, http://www.jocl.org/, [Last visited Decemeber 2012].

[12] Aparapi. API for data parallel Java, http://code.google.com/p/aparapi/, http://developer.amd.com/tools/hc/Aparapi/Pages/default.aspx, [Last visited December 2012].

[13] JogAmp JOCL. Java OpenCL, http://jogamp.org/jocl/www/, [Last visited Decemeber 2012].

[14] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo, “Generalpurpose Computation on GPUs for High Performance Cloud Computing,” *Concurrency and Computation: Practice and Experience*, vol. (in press), 2012.

[15] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter, “The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite,” Pittsburgh, PA, USA, 2010.