Runtime Translation of the Java Bytecode to OpenCL and GPU Execution of the Resulted Code

Razvan-Mihai Aciu, Horia Ciocarlie

Department of Computers, Faculty of Automation and Computers
Politehnica University Timisoara
Vasile Parvan Street, No. 2, 300223 Timisoara, Timis, Romania
razvan.aciu@cs.upt.ro, horia.ciocarlie@cs.upt.ro

Abstract: Modern GPUs provide considerable computation power, often in the range of teraflops. By using open standards such as OpenCL, which provide an abstraction layer over the GPUs physical characteristics, these can be employed to solve general computation tasks. Massively parallel algorithms used in domains such as simulation, graphics, artificial intelligence can greatly expand their application range. It is of great importance for an application to run parts of itself on GPUs and in this respect a number of issues such as OpenCL code generation, data serialization and synchronization between application and GPU must be observed. At the same time, the GPU limitations impose some limits on their applicability to general computation tasks, so an application must carefully assess what parts are suitable for this kind of execution. The computing resources must be abstracted and when possible these should be interchangeable without modifying the source code. We present a new algorithm and library which dynamically generates OpenCL code at runtime for parts of its application in order to run these parts on GPU. Our library automatically handles tasks such as data serialization and synchronization. The practical results are significant and we succeeded in obtaining important speedups using only a straightforward Java implementation of the test algorithm, without any platform specific constructs.

Keywords: Java; OpenCL; GPU; code generation

1 Introduction

For massively parallel algorithms, GPUs can offer an important speedup, often reducing their execution time by several times. In bioinformatics, using highly optimized libraries and GPU finely tuned algorithms, speedups of up to 1000x were reported [1]. Two top consumer GPUs in 2015 are AMD Radeon™ Fury X [2] with 4096 streaming cores, 4 GB memory, 8.6 TFLOPS FP32 and NVIDIA GeForce® GTX™ Titan X [3] with 3072 streaming processors, 12 GB memory and 7 TFLOPS FP32.

These GPUs are optimized especially by FP32 computing, so their FP64 performance is much lower (GeForce® GTX™ Titan X has 0.2 TFLOPS FP64), and Intel Xeon X7560 has 72.51 GFLOPS FP64 [4]. From these data it can be seen that the GPUs are valuable computing resources and their use would greatly enhance certain applications. Especially if FP32 precision is sufficient and if the algorithm is highly parallel, a single GPU can offer the performance of many desktop CPUs.

Taking into account the above considerations, it is understandable that many researchers try to develop new algorithms and frameworks capable of employing the GPU computational power. In this respect two main technologies are dominant: OpenCL and NVIDIA CUDA. We will concentrate on the OpenCL approaches, because it is a vendor neutral, open standard supported by all major vendors. Many of the aspects discussed also apply to CUDA because the physical structure of different GPUs has many common elements and both OpenCL and CUDA are abstraction layers over that physical structure.

Up to OpenCL 2.1 [5], which provides both a high level language (a subset of C++14) and a portable intermediate representation (SPIR-V), OpenCL programs were restricted to a subset of ISO C99. For maximal performance, the application interface (API) is also provided in C. The creation of an OpenCL application mainly involves the following tasks:

- creating the OpenCL code which will run on GPU,
- conversion of the application data in a format suitable for GPU execution,
- application-GPU communication and synchronization,
- data retrieval from GPU and its conversion into the application format.

If we consider a direct translation of the application code and its associated data structures to OpenCL, it becomes apparent that the above tasks are in fact standard procedures which can be automatically addressed by specialized tools or libraries. For example, the translation of the application code into OpenCL code is in fact a problem addressed in compiler theory by translators which outputs code in a high level language. The other tasks can also be automated. The resulted translation is similar with the original, in the same way a compiler creates a new representation of the original code. We do not address here the problem of creating GPU optimized code from a general algorithm, but some situations can still be optimized. A general optimization is the use of intrinsics, directly translated into native constructions for specific situations. We propose an algorithm and a library which automatically handles the above tasks, greatly simplifying the GPU/application interoperation. Our implementation employs the mentioned optimizations, translating when possible to native OpenCL operations.

The automatic translation from application to OpenCL has the advantage that it is easy to use and that it hides most of the GPU related aspects. The programmer does not need to create special data structures and he does not need explicit

serialization, deserialization and synchronization in order to communicate with the GPU. Optimized kernels or libraries can also be used if needed, or they can replace in time the automatic generated code.

2 Related Work in OpenCL Code Generation

In this section we discuss some existing libraries which automatically convert parts of their application code to OpenCL. There are several approaches in doing this and we highlight some of the benefits and drawbacks of each method. The first method is to use the application source code to generate OpenCL, by employing preprocessor commands. This approach is taken by libraries such as Bolt [6] for C++. Bolt offers macros such as BOLT_FUNCTOR which take a part of the source code and transform it to a string representation. These strings are later combined to form an OpenCL kernel. There is no further processing of the generated strings and they are used in the form they were captured, linked together with some glue code and ordered in a certain form. This method is directly applicable to languages which are supersets of the OpenCL itself, such as C/C++. There are notable advantages, one of which we should mention is the fact that the data structures have the same layout both in the host application and in the OpenCL code. The exact data layout can be enforced by using alignment specifiers. This simplifies considerably the interoperability between the host and the kernel. If the application data has a compact structure (without pointers to separately allocated structures), it can be sent in its native form to GPU so there is no serialization/deserialization overhead. At the same time, the code is the same in the application and in kernel, which simplifies debugging and CPU fallback computation when no GPU is available. This method has some weak points, among, which is the necessity for all the GPU involved code and its dependencies to adhere to the OpenCL subset. For simple kernels this is simple to accomplish, but if there are library dependencies, the entire libraries must be written in an OpenCL compatible C/C++ subset. In the source code, all the dependencies which are not included as separate files must be explicitly enclosed in specific macros. Another disadvantage is that the kernels cannot handle runtime provided code, such as plugins or short code snippets or formulas that need to be run on GPU.

A second method is for the programmer to explicitly build at runtime a representation for the needed computation and to generate the kernel from that representation. This intermediate representation can be a form of Abstract Syntax Tree (AST). By using well known algorithms, code can be generated from this AST. This approach is taken by libraries such as ArrayFire [7]. The method is suitable especially to combine library provided GPU accelerated functions into single kernels, eliminating host-GPU transfers between these functions, as in the case they were run separately. The AST leaves are the data involved in

computation, interfaced by proxy adapters. The AST nodes are functions and operators. Using operators overloading, in many cases the AST building for expressions can be syntactically abstracted as expressions written using the common infix operators, including their precedence rules and parenthesis. The method is very flexible and if the AST nodes allow (if there are nodes for loops, declarations, etc.), any construction can be generated. Runtime provided expressions or code snippets can be easily compiled by simply parsing them and constructing the associated AST. There are some drawbacks of this method such as the need to manually write the AST. For small kernels, composed from simple expression which only combine library functions, this is an easy task, but for large kernels that need declarations, decisions and loops, this task can be complex and tedious. Another disadvantage is that the AST code is completely distinct from the application code and this makes it difficult to debug and to interface with the application structures, even using predefined adapters.

Another method is to use the reflection capabilities of a programming language which enables the application to inspect its own code. This method is suitable for languages with strong reflection capabilities, such as Java and C#. Using this method, the application decompiles parts of its own code and translates them into OpenCL. The translation starts with the computation entry point (generally an interface with a method which defines the kernel body) and adds to the generated code all the required dependencies in a recursive manner. The associated data structures are also decompiled and translated. This approach is taken by libraries such as Aparapi [8]. Aparapi generates OpenCL code at runtime from an instance which implements the abstract class Kernel. The application overridden *run* method of this class is the OpenCL kernel entry point. Aparapi also serializes/deserializes the data needed for the kernel execution. This approach is very flexible and it succeeds in automating many GPU related tasks. At the same time, the GPU kernel is mirrored by the application code, which results in certain benefits such as ease of debugging (by debugging the application using standard Java tools) and the capacity to use CPU fallback computation when no GPU is available. Another benefit is the possibility of translating specific constructs into OpenCL optimized forms. Many of the Java Math class methods have corresponding OpenCL primitives and other commonly used primitives, such as the dot and cross products can be added. This method also allows plugins for GPU execution to be added at runtime. A drawback of these methods is the increased time needed to generate a kernel due to the overhead incurred by the code disassembly, but this can be alleviated by caching the generated kernel. In the particular case of Aparapi, for now it is designed as a thin layer over OpenCL, so it uses some specific functions such as *getGlobalId*. More important, it does not handle data structures which contain objects (the support is limited to single dimension arrays of primitive types), so complex data structures need to be handled manually by writing adapter code. Support for reference types is planned on certain architectures, such as the Heterogeneous System Architecture (HSA).

Another approach is the OpenJDK Sumatra [9] project with the primary goal to enable the Java applications to take advantage of the GPUs or other devices from the same class. The Sumatra project aims to convert specific constructs such as the Java 8 Stream API to an abstract representation (HSA Intermediate Language – HSAIL) from which it can be further converted to specific architectures, such as CPUs or GPUs. A difference from the above cases is that the Sumatra project tries to delegate the GPU interconnection tasks to the OpenJDK components (compiler, virtual machine), making the GPU a first class citizen of the Java supported architectures. For now the Sumatra project is in its early stages and it also depends on the success of emerging technologies such as HSA, but if it succeeds, it can be a significant achievement for the Java heterogeneous computing capabilities.

3 The Proposed Algorithm and Library

Our algorithm uses runtime reflection to access the application code, followed by disassembly, analysis and code generation steps in order to convert the relevant code to OpenCL. In the execution phase the required data is serialized automatically and transferred to GPU. The algorithm also handles the GPU synchronization and results retrieval, followed by a conversion into the application data structures. The main objective is to abstract as much as possible the GPU execution. Exactly the same code should run both on GPU and on CPU, in order to allow easy debugging and to provide CPU fallback execution when no GPU is available. Our library can generate OpenCL code for moderately complex situations such as data structures containing reference types (especially classes), exceptions handling and dynamic memory allocation. In this respect we depart essentially from libraries such as Aparapi, which are thin layers over OpenCL and in which many of the OpenCL requirements are explicitly exposed. Of course the abstraction layer can imply some performance loss. If more optimization is required, the programmer can translate some constructs into a more idiomatic code for GPU execution.

The entry point into the algorithm is a tasks scheduler which enqueues user tasks, executes them on GPU and retrieves the results. We extend [10] the standard Java thread pools with MapReduce [11] semantics and asynchronous, event driven receiving capabilities, similarly with the Node.js [12] non-blocking I/O. The library is designed in a way that allows distributed execution on many computing resources such as CPU, GPU or remote computers. When a result is received, it is passed to the *set* method of an object which implements the *Destination* interface. This method receives the resulted data and a task unique identifier such as an array index. A *Destination* can abstract over simple collections such as arrays and maps, or it can implement more complex processing, by immediately handling the data. In this way, if the specific application algorithm allows, the results can be directly used on arrival without storing them first. The scheduler starts running the

tasks as soon as they are added. In the end, after all the tasks are added, a synchronization method is called in order to ensure the end of all computations and the availability of the results.

A task is implemented as an object which implements the *Task* interface with *run* as its single method. A task encapsulates all its specific data. It is created on the application side and asynchronously enqueued for GPU execution with the scheduler *add* method. This method also associates the task with a unique identifier which will be used later when the result is processed. The task data is serialized and sent to GPU. After computation the result (returned by the *run* method) is received and converted to the application format, then sent to the scheduler associated destination. A generic use is given in Figure 1.

```
// Result is any user class; it encapsulates a task result
class ResultProcessor implements Destination<Integer, Result>{
        // set arguments: task unique identifier, received data
        @Override public void set(Integer id, Result ret){
                // process the received data
        }
}
class TaskProcessor implements Task<Integer, Result>{
        public TaskProcessor(/*input data*/){
                // task initialization on the host (application) side
        }
        @Override public Result run() throws Exception{
                // processes input data on GPU and returns it to application
        }
}
// ... entry point ...
// Scheduler instantiation
Scheduler <Integer, Result> scheduler=new Scheduler<>(new ResultProcessor());
// create all tasks and add them to scheduler
for(/*all data to be processed*/){
        TaskProcessor task=new TaskProcessor(/*specific initialization data*/);
        scheduler.add(id, task);   // asynchronous addition of the task to scheduler
}
scheduler.waitForAll();            // wait for all tasks to be completed
```

Figure 1
A generic use of the library

For simple cases such as the processing of Java standard collections, predefined destinations are provided so there is no need to implement the *Destination* interface. It can be seen that the processing resource is fully abstracted. Inside the *Task* implementation or on scheduler there are no OpenCL specific instructions,

but only general concepts are used, such as the task unique identifier. These allow a better representation of the application domain and a better abstraction of the computing resource. More specific instructions are given only if necessary, such as the case when multiple computing resources are present and a specific one needs to be selected.

When a new task is added to scheduler, first the scheduler checks if it already has the generated OpenCL code for that task. If the generated code exists, it is used else the task code is retrieved via reflection, disassembled and its corresponding OpenCL code is generated. The task instances are also serialized and in order to do this, their fields are investigated using reflection and the content is put in a memory buffer which will become the kernels global heap. Both code and data disassembly and serialization are recursive operations and the process ends when all the dependencies were processed. After the code is run on GPU, the heap is retrieved and its content is deserialized into the application data structures. This step updates in application the data modified by the kernel and it makes available the kernel allocated structures.

3.1 Data Serialization and Retrieval

The serialized data and the heap space for dynamic memory allocation are provided in a single buffer referred as the OpenCL global memory space or heap. All data references are converted in offsets in this heap. The kernel accesses data with a heap based indexing, similar with an array access. This method has a performance impact if the GPU native code does not support indexed access. In this case, a separate instruction must add the offset to the heap base pointer. Approaches such as the Sumatra project can use the shared virtual memory (SVM) feature from OpenCL 2.0, because they are tied to a specific Java Virtual Machine (JVM) implementation (OpenJDK), but in the general case JVM does not enforce a memory layout for many of its data structures, so we cannot directly use SVM.

The serialization algorithm uses the JVM primitive values as is. The only mention here is that the host and the GPU endianness must be observed. The reference values are split into two cases: class instances and arrays. In both cases the first member is a unique identifier for that class or array type. For class instances all the primitive values and references are added in the order given by reflection. Each class is implemented as an OpenCL structure and all the class instances are created and accessed through this structure. For arrays the array length follows and then the elements. The arrays for primitives are separate types and the arrays for references are implemented as a single type of array of *Object*, using type erasure. In order to ensure that the offsets of the instance members or array elements are known when the instance itself is serialized, these are serialized first, using a recursive process. The static members are implemented as a part of the global context and they also are stored on heap.

3.2 Code Generation

JVM has some features which are not available in OpenCL, such as dynamic allocation, exceptions throwing/catching, virtual method calls and recursion. These features need to be implemented as a library on top of the OpenCL primitives. Other features such as calling host functions for now are unavailable in OpenCL, so there is no method in doing this other than stopping the kernel, making that call from inside the application and restarting the kernel. This forbids the use of I/O functions such as the file or network API.

In the first translation step the JVM bytecode of each method is simulated linearly (without looping or branching) using a symbolic stack. A cell in this stack is an Abstract Syntax Tree (AST) node and each bytecode which influences the stack will combine these nodes. For example a numeric constant push bytecode creates an AST leaf for a constant value and an addition bytecode creates a new addition AST node from the top two AST nodes on stack. Other nodes are created or combined by instructions which do not operate on stack for example, the *goto* opcode. After this step we have a full AST for each method. We traverse this AST in order to generate the OpenCL code.

Memory allocation is a complex topic and it is particularly important for languages with automatic memory allocation, especially if there are no value semantics for classes, so all class instances need to be dynamically allocated (if the allocation is not optimized by the compiler). On massively threaded environments such as the ones provided by GPUs, where several thousands of tasks can run concurrently, special designed memory allocators are strongly required, or else they will become a performance bottleneck [13]. In our implementation we used a lightweight memory allocator, similar with [14], which is capable for now only to allocate data. This approach ensures a high allocation throughput (only one atomic operation is used to serialize the heap access) and no memory overhead for the allocated blocks, but it does not reclaim the unused memory, so the programmer must be careful with the number of allocations. For Java applications with intensive memory allocation this can be a problem and probably our allocator must be extended with full garbage collecting capabilities. On the programmer side, taking into account that the cases when GPU execution is required are especially the cases when a high performance is needed, simple methods can be used to minimize memory allocations and to optimize the application both for GPU and for CPU by reducing the pressure on the memory allocator. Our own interface library for accessing native OpenCL functionality reduces memory allocations by reusing the existent objects. In order to accomplish this, we devised the object based operations, such as the vectors addition to operate on the first argument instead of creating a new result vector. Because in OpenCL there are no global variables, each function needs to receive the global state (in our case the heap base address) as a supplementary argument. This incurs no overhead on the function call because the OpenCL compiler is required to inline all the functions so no code is generated for parameters passing, but only for their actual use.

Even in version 2.1 (which is based on a subset of C++), OpenCL does not provide exceptions handling. Even if a programmer does not throw exceptions in the code designed for GPU execution, these exceptions may appear from the memory allocator if it runs out of memory. Because there is no support for facilities such as stack unwinding, we implemented the exceptions as the functions return values. In this respect every function returns an integer value, which is an offset on heap. If that value is 0 (corresponding to a Java null pointer), no exception was generated. If an exception is generated, the exception object is allocated on heap and its index is returned. Every function call is guarded for non-zero return values and if this case happens, the enclosing function immediately exits, propagating further the received exception. If the exception is not handled, the kernel will be terminated and the exception object will be passed to application as a result for that particular task. An *OutOfMemoryError* object is preallocated for out of memory cases. Because the functions returns are used for exception propagation, if the function needs to return a value, it is returned through a supplementary reference parameter added to that function. This parameter holds the address of the variable which will receive the returned value. Using this parameter, the *return* instruction stores its expression into the provided address. We analyzed the GPU native code resulted from the OpenCL code in order to evaluate the performance impact of this design decision. Because the OpenCL compiler is required to inline all its functions, we saw that the pointer indirection used for storing the return value was simply optimized away and a direct store was used instead, so there is no performance loss using this model. In the same way, the guard for non-zero returned values and early enclosing function exit were propagated to the origin point (in our case the memory allocator) and an immediate kernel exit is performed if the subsequent code does not contain *try…catch* clauses, so this checking was also optimized away.

Some core Java classes such as *Object*, *Integer* and *Math* are treated as intrinsics. This allows a better code generation, optimized to use the OpenCL predefined functions. Many *Math* functions are already defined in OpenCL, so these can be translated directly into native functions. Other OpenCL available functions, such as the dot and cross products are provided in an auxiliary math library, which is also treated as an intrinsic. If the code is not executed on GPU, this library automatically uses a regular Java implementation for its functions. As an example of code generation, in Figure 2 we present the Java method we implemented to compute a Mandelbrot point and in Figure 3 (augmented with some comments and formatted to reduce the number of lines) we present its OpenCL generated code.

In order to be able to generate code for overloaded functions, we devised a new name mangling system because the JVM system uses characters which do not translate in valid C identifiers. We needed also to differentiate between functions with the same name from different classes, so in our system we append to a function name both its context (package and class name) and its signature.

```
int mandelbrot(float x,float y){
        final int ITERS=256;
        float xi=0,yi=0;
        float xtemp;
        int iteration;
        x=translate(x, 1.2501276f, 3.030971f, 0.31972017f, 0.34425741f);
        y=translate(y, -2.9956186f, 1.8466532f, 0.03119091f, 0.0572281593f);
        for(iteration=0;xi*xi+yi*yi<2*2 && iteration<ITERS;iteration++){
                xtemp=xi*xi-yi*yi+x;
                yi=2*xi*yi+y;
                xi=xtemp;
                }
        return iteration;
}
```

Figure 2
The original Java version of the Mandelbrot method

OpenCL does not have pointers to functions or other indirect calls. The virtual functions which regularly are implemented using virtual tables with pointers to the function implementation need to be implemented in other way. In our implementation the unique class identifier, which is the first member of any class structure, can be used for this purpose. This identifier is not an index in heap, but it is an index into a host maintained vector with the structures of the generated classes, so for a specific kernel it maintains its value regardless the actual data serialized in heap. Using this id the virtual functions can be implemented by testing the current object id in a *switch* instruction with dispatches for all possible situations (for all classes from a specific hierarchy which provide a specific implementation of that function). In the same way can be implemented dynamic dispatch for all the implemented interfaces. For now our generator does not fully implement dynamic dispatch (virtual methods), so we can use only the Java final classes, on which the specific called method can be inferred by the compiler.

Because the GPUs do not provide an execution stack, recursive functions are not implicitly supported [15]. The programmer must provide an iterative version of the algorithm or the recursion must be implemented using an explicit stack. In this version our library does not support code generation for recursive functions. We consider this a limitation which can be solved and we are trying to address it in the next version.

```
// idxtype is the integer type needed to access the heap vector (unsigned int)
// _g is a pointer to heap
// _l is a pointer to a location where the function return value will be stored
// return: 0 if no exceptions occured, or a heap index for the exception object
idxtype    tests_D_T3Work_D_mandelbrot_LP_FF_RP_I
                                        (_G _g, idxtype _this, float x, float y, int *_1){
int        ITERS, iteration, _TV3;              // _TV* are temporary variables
float      yi, xi, xtemp, _TV0, _TV1, _TV2;
idxtype _0;                  // used to test if exceptions occured and propagate them
ITERS=256;
xi=0.0; yi=0.0;
// each function call is guarded against exceptions
if((_0=tests_D_T3Work_D_translate_LP_FFFFF_RP_F(_g, _this, x,
          1.2501276,3.030971, 0.31972017,0.34425741, &_TV0))!=0)return _0;
x=_TV0;
if((_0=tests_D_T3Work_D_translate_LP_FFFFF_RP_F(_g, _this, y, -2.9956186,
          1.8466532, 0.03119091, 0.0572281593, &_TV0))!=0)return _0;
y=_TV0;
iteration=0;
goto _TMP172;
_TMP173:;
_TV0=xi*xi;_TV1=yi*yi;_TV2=_TV0-_TV1;_TV0=_TV2+x;xtemp=_TV0;
_TV0=2.0*xi;_TV1=_TV0*yi;_TV0=_TV1+y;yi=_TV0;
xi=xtemp;_TV3=iteration+1;iteration=_TV3;
_TMP172:;
_TV0=xi*xi;_TV1=yi*yi;_TV2=_TV0+_TV1;
// FCMPG is a macro which implement the JVM FCMPG opcode
_TV3=FCMPG((float)_TV2,(float)4.0);
_TV0=_TV3>=0;
if(_TV0)goto _TMP177;
_TV3=iteration<256;
if(_TV3)goto _TMP173;
_TMP177:;
*_1=iteration;               // setup the return value
return 0;                    // return without exceptions
}
```

Figure 3
The OpenCL generated code for the Mandelbrot method

4 Practical Results

In order to test our algorithm and library, we created a Java application which renders an image through ray casting. Primary rays are sent, without future reflections or refractions. The scene is composed of 1301 spheres. On each sphere the Mandelbrot fractal is applied as a procedural texture, so each pixel is directly computed when needed, without applying any precomputed bitmap. Besides the procedural texture, for each point is applied a simple illumination model which takes into account the angle of intersection between the ray and the sphere normal on the intersection point. Each horizontal line is considered a separate task (a work-item). The final result is shown in Figure 4. We used different Java features such as classes, members of reference types, static members. All calculations were done in FP32. Java Math library offers mostly FP64 operations, so we wrote a wrapper around these, to convert them to FP32. Because of this some operations such as *sin()* and *cos()* are executed as FP64 on CPU and FP32 on GPU. We allowed this because we assumed that in a real-world scenario when FP32 data is used, the programmer does not want to convert it to FP64 when using *sin()* and *cos()*, but if possible he will use FP32 operations. On the OpenCL side all these operations are translated into native functions. We did not use OpenCL or CPU specific features and the application ran without any modifications on CPU and on GPU. In this version our library uses for Java bytecode manipulation the ASM v5.0.3 library [16]. For standard OpenCL bindings we use JOCL v0.2.0-RC [17].

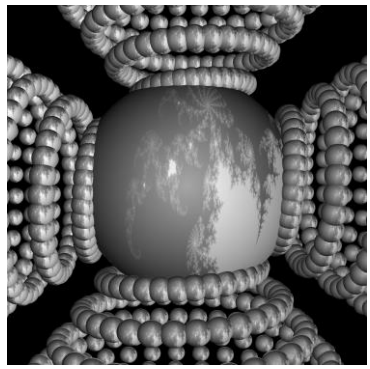

Figure 4
The result of the test program

We used the following test configuration:

- a computer with Intel® Core™ i5-3470 CPU at 3.20 GHz with 8 GB RAM, Windows 7 Home Premium SP1 on 64 bits and Java SE 8u45. This CPU has 4 cores.
- AMD Radeon™ R9 390X at 1060 MHz with 8 GB GDDR5 RAM. This GPU has 2816 streaming cores and 44 compute units.

We studied three main aspects: how the GPU execution compares with the CPU execution, how the GPU handles different workloads and how our library compares with the Aparapi library. For the comparison between GPU and CPU we used square images and we increased linearly the number of pixels in order to determine the best execution method for different sizes. Each test was run 5 times and the average value was taken. The GPU time included all the implied times: kernel generation and compilation, serialization/deserialization and the execution time. Data transfer between CPU and GPU are included in the execution time. This total time is needed only for the worst GPU case, when the computation is run only once. If the computation is run multiple times, the generated kernel can be cached and reused, so the generation and compilation time become insignificant. We measured these times across the test range and the results are given in Table 1.

Table 1
Setup times and data sizes for GPU

	10 KPixels image			43000 KPixels Image		
	Time (ms)	Heap data (KB)	Heap total (KB)	Time (ms)	Heap data (KB)	Heap total (KB)
Kernel generation	22	62.8	111	23.3	44442	45490
Kernel compilation	224			228.5		
Serialization	11.3			55.4		
Deserialization	0.65			100.5		

As expected, the kernel generation and compilation is invariant with the workload because the processed bytecode is the same. The serialization and deserialization times increase when more data need to be processed. For small workloads the result is given in Figure 5. Data was collected in increments of 10 KPixels (KP).

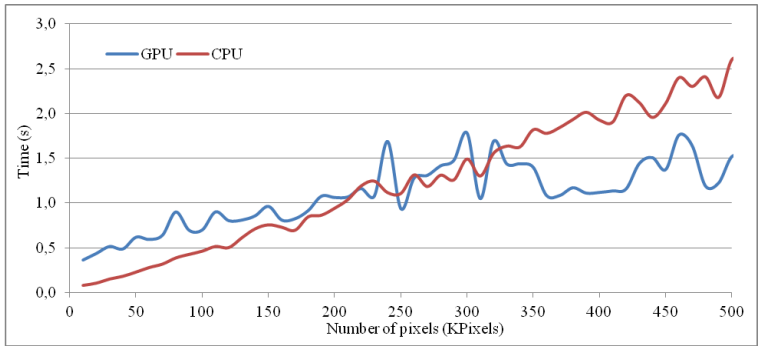

Figure 5
GPU vs. CPU execution for small workloads

It can be seen that the GPU execution starts to be faster at around 250 KP. If we subtract the time needed to generate and compile the kernel, this threshold lowers to around 100 KP. For this test application, the lower number of pixels is a GPU worst case because a square image of 100 KP has a height of around 316 pixels. It means that at maximum only 316 GPU streaming cores are working from the available 2816 streaming cores. Probably one of the few good things in this case is that a compute unit runs simultaneously fewer work-items, so the execution divergence is lower.

Next we compared the GPU vs. CPU execution across the entire possible range. On our system a single GPU execution is limited to about 30 seconds. After that, the operating system resets the graphic driver because it considers it unresponsive. On other systems this timeout can be lower, so care must be taken with long running kernels. The results are shown in Figure 6. Data was collected in increments of 1 MPixel.

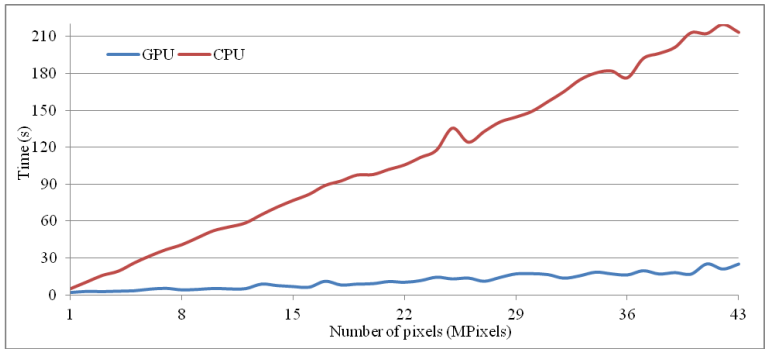

Figure 6
GPU vs. CPU execution across the entire test domain

It can be seen that when the quantity of work for a task (the image width) and also the number of tasks (the image height) increases, the GPU outruns the CPU with a linearly larger time. Because we increased the number of pixels linearly, the quantity of work also increases linearly. In this case, the difference in the number of the GPU streaming cores (2816) and the CPU cores (4) becomes apparent because in each case the increased workload is divided between all the available processing elements, so each CPU core will receive a greater amount of the increase. Even if both times increases are mostly linear, the CPU time increase is steeper than the GPU time. The maximum GPU speedup over CPU was 12.65x.

Another aspect we tested is how the GPU handles different types of workloads. For this test we kept the task size constant (the image width) and varied the number of tasks (the image height). We ran this test only on GPU and we measured the times for 3 different task sizes. The results are shown in Figure 7. Data was collected in increments of 64 tasks. We explain this graph by a combination of two factors. The general shape of each line is given by how the

workload fits in the GPU cache memory. It can be seen that for smaller workloads (width=1000), the increase is approximately linear over the entire domain. When we increase the workload, so more memory is needed, the number of cache misses increases and this increase becomes more apparent on the right side of the graph, where it strongly influences the time growth. The second factor which influences this graph is how the tasks are allocated for execution on GPU. If we add tasks so the total number is smaller than the number of the streaming cores, then the time increase for each added task is very small because all these tasks will be ran on the same batch. If the number of tasks is greater than the number of the streaming cores, we can have two extreme situations: all the tasks of a batch end approximately at the same time and in this case we have a local minimum, or when we increase the tasks by a small number the newly added tasks require a new batch only for them and in this case we have a local maximum. The situation is more complex due to the execution divergence and because not all the tasks require the same amount of work.

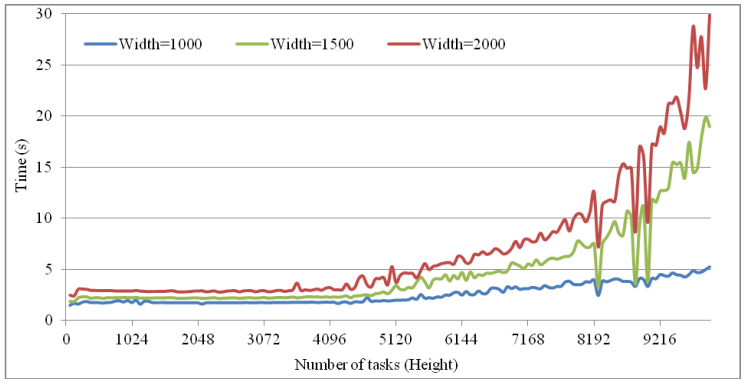

Figure 7
GPU execution for different number of tasks

A result of this analysis is that for long tasks it is better to reorganize them in such way that the amount of memory required by a task is as small as possible (or so they have common data). This optimizes the execution time by reducing the cache misses and also avoids the operating system timeout for GPU execution. The number of tasks sent for a single GPU execution can be set from the scheduler.

To compare our library with Aparapi, it was necessary to rewrite the test application into a representation suitable for Aparapi. We needed to replace some higher level data structures with lower level representations, such as:

- Aparapi does not support reference types (a part of unidimensional arrays of primitive types), so any classes used (such as Point, Line, …) were replaced with vectors of floats. For example, a Point was represented as a vector of 3 floats and a Line as a vector of 6 floats.

- Aparapi uses a single global space for all threads and the distinction between the threads data can be made using functions such as getGlobalId(). We needed to use combined vectors for all threads data and to access specific data based on the thread global id.
- Aparapi does not allow dynamic memory allocations so we needed to use global data structures to keep the returned functions values when these values are not primitive types. The thread global id was used to differentiate the threads data.

To illustrate some of these changes we show in Figure 8 how some parts of the test application are implemented using our library and in Figure 9 how these are implemented using Aparapi.

```
public class Point{
        public float x,y,z;
        public Point(){}
        public Point(float x,float y,float z){this.x=x;this.y=y;this.z=z;}
        public void set(float x,float y,float z){this.x=x;this.y=y;this.z=z;}
        public void set(Point p){x=p.x;y=p.y;z=p.z;}
        public float len(){return Math3D.length(x,y,z);}
        public void vectorFromPoints(Point origin,Point destination) {
                x=destination.x-origin.x;
                y=destination.y-origin.y;
                z=destination.z-origin.z;
        }
…
}
…
Point intersToRayOrigin=new Point();
Point centerToInters=new Point();
float     propagateRay(Line ray){…}
```

Figure 8
Part of the test application implemented with our library

Since the Aparapi version cannot use Java fundamental idioms such as classes, an algorithm written using this library needs additional proxy code to integrate with the rest of the application: translation from classes to vectors and back, merging of individual tasks data into single structures, etc.

In order to compare the execution of the Aparapi with our library, we varied both the number of pixels and the amount of work needed for each pixel. The latter measurement was needed in order to evaluate different workloads by keeping constant the amount of memory used and the number of threads. We recomputed each pixel a number of times (n), starting with n=1 (normal case). This process does not involve memory allocations.

```
final float pointLen(float []p){
        int offset = getGlobalId()*3;
        return length(p[offset],p[offset+1],p[offset+2]);
}
final void vectorFromPoints(float []dst,float []origin,float []destination){
        int offset = getGlobalId()*3;
        dst[offset]=destination[offset]-origin[offset];
        dst[offset+1]=destination[offset+1]-origin[offset+1];
        dst[offset+2]=destination[offset+2]-origin[offset+2];
}
…
final float []intersToRayOrigin;
final float []centerToInters;
final float        propagateRay(float []ray){…}
```

Figure 9
Part of the test application implemented for Aparapi

In Figure 10 we showed the results of the Aparapi executions and in Figure 11 the results from our library. For both figures data was collected in increments of 1 MPixel.

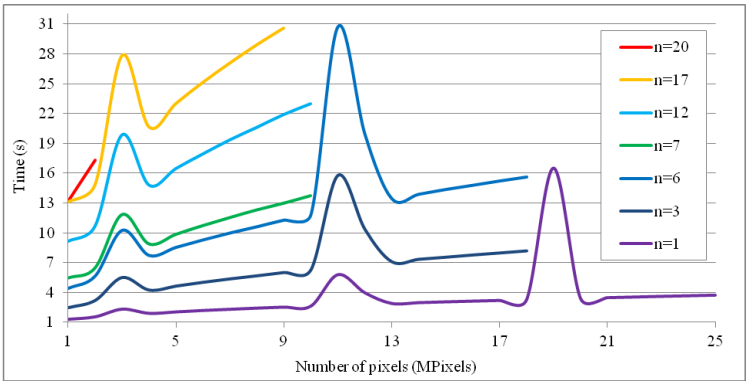

Figure 10
Aparapi execution for different number of tasks and recomputations

On the Aparapi version, the OS started to reset the graphic driver at around 25 MPixels. Our library was able to produce results up to 46 MPixels. Both libraries show approximately linear progressions, with some prominent peaks. Aparapi shows a better time and a smaller grow angle. Our library shows more irregularities from an ideal linear progression and, as discussed previously, we

consider this a combined effect of the data layout and the GPU cache. Aparapi has smaller irregularities because in its case the data are already vectorized and ordered by tasks, which improves the data locality. In both implementations the maximum running time, after which the OS started to reset the graphic driver was of about 31 seconds. When the peak would exceed 31 seconds, the application would crash.

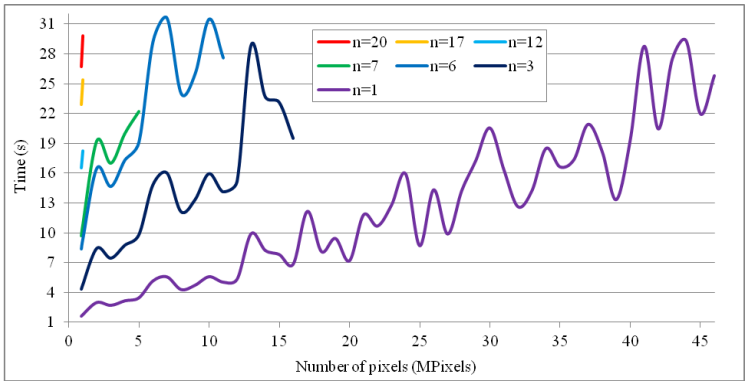

Figure 11
Our library execution for different number of tasks and recomputations

Conclusions

In this paper we proposed an algorithm and library which enable Java execution on GPUs. We used reflection in order to access the application code and OpenCL code generation to create the GPU kernels. This approach is also suitable for runtime provided code such as plugins. Our library automatically handles tasks such as data serialization/deserialization, GPU communication and synchronization. The data serialization system can process complex data structures, which enables the application to use for GPU execution classes, reference type fields, any type of arrays and static methods.

The library defines a Java compatibility layer over the OpenCL standard. This layer allows us to use exceptions handling and dynamic memory allocation. In the future we hope to extend it with virtual methods calls (dynamic dispatch) and recursive calls. Where possible, the OpenCL native functions are used instead of the Java standard libraries. We also provided an auxiliary library for the OpenCL primitives which do not have a correspondent into Java standard libraries. This library is portable, so it can be used both on GPU and on CPU.

Our algorithm uses a MapReduce model to manage the parallel tasks. The tasks return values are directly sent to a handler. In some cases this allows the results processing on arrival, without needing to store them. The tasks creation and management is abstracted over the computing resources, so the code can be executed both on GPU and on CPU without any modifications. This simplifies the

code maintenance, allows for an easy debugging of the code and the CPU can be used as a fallback resource when no suitable GPU is available.

We tested our library using a test application written in standard Java code, without any OpenCL specific constructs. For our test configuration we obtained significant speedups of up to 12.65x on GPU over the CPU execution. We consider the most important conclusion of this research the fact that parts of standard Java applications which use classes, dynamic memory allocation and exceptions handling (but for now without virtual calls and recursion) can be automatically translated to OpenCL and run on GPU and this can bring certain advantages. Our algorithm and library provides the capability to write Java code which can be easily integrated with complex data structures, code which does not need platform specific calls. This greatly simplifies the goal of running complex applications on different computing resources such as CPU or GPU. This is an important achievement over existing libraries such as Aparapi, which were designed as thin layers over OpenCL and requires the programmer to use specific OpenCL calls. In our tests Aparapi obtained a better time than our library, but it was capable of handling only a restricted domain of the test data and it required coding the application in a manner, which is not specific to Java (for example without classes) and which requires proxy code to translate the application data to a suitable representation and back. Our future research will concentrate on increasing the range of the applications which can run on GPU, develop better optimizations and obtain an increased reliability for GPU execution.

Acknowledgement

This work was partially supported by the strategic grant POSDRU/159/1.5/S/137070 (2014) of the Ministry of National Education, Romania, co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

References

[1] Lorenzo Dematté , Davide Prandi: GPU Computing for Systems Biology, Briefings in Bioinformatics, Vol. 11, No. 3, pp. 323-333, 2010

[2] AMD: A New Era in PC Gaming, E3, Los Angeles, California, U.S., 2015

[3] Jen-Hsun Huang: Opening Keynote, GPU Technology Conference, San Jose, California, U.S., 2015

[4] Paweł Gepner, David L. Fraser, Michał F. Kowalik, Kazimierz Waćkowski: Evaluating New Architectural Features of the Intel(R) Xeon(R) 7500 Processor for HPC Workloads, Computer Science, Vol 12, 2011

[5] Khronos Group: The Open Standard for Parallel Programming of Heterogeneous Systems, https://www.khronos.org/opencl/, download time: 14.07.2015

[6] AMD: Bolt, https://github.com/HSA-Libraries/Bolt, download time: 16.07.2015

[7] Kyle Spafford: ArrayFire: A Productive GPU Software Library for Defense and Intelligence Applications, GPU Technology Conference, San Jose, California, U.S., 2013

[8] AMD: Aparapi, https://github.com/aparapi/aparapi, download time: 16.07.2015

[9] Eric Caspole: OpenJDK Sumatra Project: Bringing the GPU to Java, AMD Developer Summit (APU13), 2013

[10] Razvan-Mihai Aciu, Horia Ciocarlie: Framework for the Distributed Computing of the Application Components, Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, 2014, Brunów, Poland, Springer, ISBN: 978-3-319-07012-4

[11] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, Commun. ACM 51, 2008

[12] Mike Cantelon, Marc Harter, T. J. Holowaychuk, Nathan Rajlich: Node.js in Action, Manning, 2014, ISBN 9781617290572

[13] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Dieter Schmalstieg: ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU, Innovative Parallel Computing (InPar), San Jose, California, U.S., 2012

[14] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, Haibo Lin: MapCG: Writing Parallel Program Portable between CPU and GPU, Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, Vienna, Austria, 2010

[15] Ke Yang, Bingsheng He, Qiong Luo, Pedro V. Sander, Jiaoying Shi: Stack-Based Parallel Recursion on Graphics Processors, ACM Sigplan Notices (Vol. 44, No. 4, pp. 299-300), 2009

[16] Eugene Kuleshov: Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns, Sixth International Conference on Aspect-Oriented Software Development, Vancouver, British Columbia, Canada, 2007

[17] Java bindings for OpenCL, http://www.jocl.org/, download time: 19.07.2015