An AO system for OO-GPU programming

Andrea Fornaia, Christian Napoli, Giuseppe Pappalardo, and Emiliano Tramontana
Department of Mathematics and Informatics
University of Catania, Viale A. Doria 6, 95125 Catania, Italy
{fornaia, napoli, pappalardo, tramontana}@dmi.unict.it

Abstract—**Recent technologies, like general purpose computing GPU, have a major limitation consisting in the difficulties that developers face when implementing parallel code using device-oriented languages. This paper aims to assist developers by automatically producing snippets of code handling GPU-oriented tasks. Our proposed approach is based on Aspect-Oriented-Programming and generates modules in CUDA C compliant code, which are encapsulated and connected by means of JNI. By means of a set of predefined functions we separate the application code from device-dependent concerns, including device memory allocation and management. Moreover, bandwidth utilisation and cores occupancy is automatically handled in order to minimise the overhead caused by host to device communications and the computational imbalance, which often tampers with the effective speedup of a GPU parallelised code.**

Keywords—Code generation, GPU programming, separation of concerns.

I. INTRODUCTION

For device-oriented parallel code, such as distributed high performance computing (HPC) and hardware-dependent paradigms, developers have an additional task when building an application which is taking into account the physical structure of the host (or network) [1], [2]. Developers have to consider parallelisation and communication overhead, the required bandwidth etc. [3]. Therefore, developers strive to achieve in their solutions both flexibility and high modularity. This results in increased development time and costs, sometimes with a low-performing code. Moreover, current development tools do not offer a sufficient abstraction level, instead provide a low degree of modularity to applications [4]. Furthermore, developers should take into account very complex scenarios in order to parameterise their code e.g. for different sizes for the solution domains, different number of threads and block, different data sizes, and therefore different solutions to obtain maximum core occupancy and transfer bandwidth. As a result it is difficult to separate the different concerns, overcharging the developer (and the code) with the handling of a multitude of responsibilities [5].

This paper proposes a new paradigmatic solution letting developers that use object-oriented (OO) code to develop GPU-specific code or low-level device-oriented code by means of a friendly toolbox. This toolbox uses cooperating agents to assist the development of scalable modular code that take advantage of GPU devices, freeing developers from the need to handle a device oriented language. It also provides the management of memory allocations and overall communications between host and computing device. Aspect-oriented programming [6], [7] is used as a glue to enhance an application with environment-specific choices, such as the selection of a specific task-driven code at runtime. Therefore, our proposed approach brings a substantial improvement in terms of modularity, performance, reusability of code and separation of concerns [8], [9], [10], [11], [12].

The developed toolbox provides enhanced reusability for parallel computation of previously written code (both object based or device oriented), by using several agents which interprete the behaviour of the OO code and uses a dedicated translation utility [13], [14]. Tools such as LIME [15] or AeminimumGPU [16] derive a customised language and a run-time environment, however require specific compilers and force developers to use a non-standard programming language while giving no options for standard code reusability (it is impossible for such paradigms to take advantage of reused sequential OO code and obtain parallel versions).

While OpenCL provides developers with fine-grain control of host and kernel code, the handling of low-level details is a significant overhead for the developer. The proposed toolbox, instead, requires no knowledge of the device-oriented language, instead the developer writes standard OO code, and takes care of connecting the toolbox by using some annotations.

Recent works, such as [17], have partially automated several processes in the field of code control to avoid conflicts or misleading behaviours, but even in this case it is ultimately the programmer's responsibility to structure their codes in the appropriate manner. An approach has been derived to mechanically determine how a program accesses data [18], and, other analysers have focused on extracting the structure of a software system to determine some structural properties [19], [20], [21], [22], [23], [24], [25], [26], [27]. Such analyses are paramount for assessing the possibility to transform a program in such a way to have parallelism while avoiding data inconsistency. In [28] a Java software system has been presented, based on an approach that derives an entirely new set of syntactical rules for the use of a proprietary meta-compiler. As far as our understanding, no significative further step has been made towards an high-level and self contained toolbox for easy development of GPU oriented software within an OO paradigm.

II. GPGPU AND CUDA PROGRAMMING

GPU programmers have to consider the underlying hardware in order to write any GPU-enabled code (now on simply GPU *kernel*). Graphics processors provide a big number of simple multithreaded cores offering the potential for dramatic speedups for a variety of general purpose applications when compared to the CPU sequential computation [29], [30], [31], [32], [33].

The launch of the Nvdia™ CUDA technology has opened a new era for GPGPU computing allowing the design and implementation of parallel GPU oriented algorithms without any knowledge of OpenGL, DirectX or the graphics pipeline. A CUDA-enabled GPU is composed of several MIMD (multiple instruction multiple data) multiprocessors that contain a set of SIMD processors (single instruction single data). Each multiprocessor has a shared memory that can be accessed from each of its processors, and also shares a bigger global memory common to all the multiprocessors. Basically, a CUDA kernel makes use of threading between the SIMD processors, where a single computation is performed [3]. Moreover, the GPU card allows an advanced geometrical enumeration for threads described by a 3–dimensional structure for the 3 spatial axis (even if the z axis is actually only a logical extension) [5]. Furthermore, it is possible to collect a set of threads in logical 3–dimensional blocks that are executed on the same multiprocessor.

In CUDA programming model, an application consists of a *host* program that executes on the CPU and other parallel *kernel* programs executing on the GPU [34], [35]. A *kernel* program is executed by a set of parallel threads. The *host* program can dynamically allocate device *global* memory to the GPU and copy data to (and from) such a memory from (and to) the memory on the CPU. Moreover, the *host* program can dynamically set the number of threads that run on a *kernel* program. Threads are organised in blocks, and each block has its own *shared* memory, which can be accessed only by each thread on the same block.

It is paramount that interactions between CPU and GPU are minimised, this avoids communication bottlenecks and delays due to data transfers. Necessary data transfers should try to maximise the bandwidth usage, i.e. CPU and GPU perform as least as possible interactions and transfer a large amount of data each time.

A. Bandwidth measurements

The Bandwidth is indeed one of the most important factors for performance. The best practice in CUDA C programming recommends that almost all GPU adaptation changes to code should be made in the context of how they affect bandwidth.

Bandwidth can be dramatically affected by the choice of memory in which data is stored, how the data is laid out and the order in which it is accessed, as well as other factors due to the computation itself. In order to obtain an accurate estimate of the possible performances it is required to calculate the effective bandwidth which, generally, strongly differs from the theoretical bandwidth (the latter is much greater than the former). The theoretical maximum bandwidth B_{TH} is

$$B_{\text{TH}} = n_{\text{M}} C_{\text{M}} R_{\text{M}} \tag{1}$$

where C_{M} is the maximum memory clock, n_{M} is the number of bits of the memory interface, and R_{M} is the memory data rate (1 if single rate, 2 if double rate, etc.). Moreover, to obtain an accurate estimate of the effective bandwidth B_{EFF}, such computations should be performed at execution time by means of the following equation

$$B_{\text{EFF}} = \frac{(B_{\text{R}} + B_{\text{W}})}{t} \tag{2}$$

where B_{R} is the number of bytes read per kernel, B_{W} is the number of bytes written per kernel, and t is the time. On the other hand, B_{EFF} cannot be computed before hand, but only after observing the runtime execution. The presented solution enable us to perform these operations and to obtain a real time estimate of the bandwidth occupancy rateo

$$B_{\text{OR}} = \frac{B_{\text{EFF}}}{B_{\text{TH}}} \tag{3}$$

At runtime it could be useful to compute B_{EFF} as

$$B_{\text{EFF}} = n_{\text{O}} \left(\prod_{i=1}^{D} l_i \right) \texttt{sizeof(}\textit{TYPE}\texttt{)} \tag{4}$$

where n_{O} is the number of operations (e.g. 2 for read and write), D the maximum number of dimensions of the data structure in transfer, l_i the length along the i-esime dimension, and `sizeof(`*TYPE*`)` the dimension in bytes of one unit of data for the specified type. Therefore it follows that

$$B_{\text{OR}} = \frac{n_{\text{O}}}{n_{\text{M}} C_{\text{M}} R_{\text{M}}} \left(\prod_{i=1}^{D} l_i \right) \texttt{sizeof(}\textit{TYPE}\texttt{)} \tag{5}$$

Since we are interested in host to device transfers (and vice versa) then $n_{\text{O}} = 2$, moreover C_{M} and R_{M} depend on the hardware, and they are un-mutable during the execution. Therefore, given a fixed constant

$$K_{\text{HW}} = \frac{n_{\text{O}}}{n_{\text{M}} C_{\text{M}} R_{\text{M}}} \tag{6}$$

it follows that

$$B_{\text{OR}} = K_{\text{HW}} \left(\prod_{i=1}^{D} l_i \right) \texttt{sizeof(}\textit{TYPE}\texttt{)} \tag{7}$$

This latter gives the exact bandwidth usage.

B. Memory optimisations

While the bandwidth occupancy rateo give us an estimate of the performance of the code, in order to improve such performances a major part of the effort should be directed toward memory optimisations. A performant code should indeed maximise the bandwidth occupancy rateo, but such a bandwidth is best served by using as much fast memory and as little slow-access memory as possible (this practice applies both to the device memory and to the host memory).

In order to gain performance, it is important to reduce the number of data transfers between host and device, sometimes also by running directly on the GPU portions of serial code (or portions of code the CPU could easily outperform).

For the same reason data structures could be created both in the device and in the host in order to serve as an intermediate buffer. Such a buffer could also be useful to avoid small transfers, organising larger transfers which should perform better even in case of non-contiguous regions of memory (these would be packed in an unique compact buffer and then unpacked at their destination).

A major improvement in memory usage is finally granted by using page-locked memory, also known as *pinned* memory. By using the pinned memory the bandwidth usage should be maximised (hence, limiting to the transfers between host and device). In order to use the pinned memory the CUDA libraries provide the `cudaHostAlloc()` and `cudaHostRegister()` functions: the first allocates region of memory in pinned modality, while the latter is used to pin the memory on the fly without allocating a separate buffer.

While the use of pinned memory could improve the performance, this practice is likely to be difficult for the developers, who risk to take on too much responsibilities. Moreover, the usage of pinned memory does not give a general solution for every code since pinned memory is a rare resource and an excessive use could end up in an overall reduction of the system performances. Finally, memory paging is often a heavyweight operation when compared to normal memory management. This results in a trade-off situation which should be carefully analysed before taking any action. The proposed solution is intended to spare the developers from such concerns by taking care of this issue with automatic evaluations and countermeasures.

C. Asynchronous transfers

In a standard situation the developer may decide to transfer data between host and device by means of the `cudaMemcpy()` function, which is a blocking transfer. In other words, such an operation constitutes a barrier and returns the control to the thread only after the entire data transfer is completed.

CUDA architecture offers a different solution for memory transfer by means of the `cudaMemcpyAsync()` function which is a non-blocking variant of the previous one. This function returns control immediately with the related consequences. Moreover, this function requires the use of the pinned memory, and, for security reasons, to use the so called *streams*. A stream is a sequence of operations that are performed on the device following a certain order. Streams must be properly used while using asynchronous transfers in order to correctly access data only after they have been transferred. On the other hand different streams can be overlapped. Asynchronous transfers enable us to overlap data transfers with computations, therefore their proper use could tremendously increase performances, however they could be very tricky for the developer, and again assistance could be required. Also this kind of assistance is provided by our developed solution.

D. Cores occupancy

Another key point in order to maximise the GPU performances is the core occupancy. While a task should run unconstrained, its workload should be correctly designed so as to take advantage of a number of threads that exactly matches the number of available GPU cores. The best practice recommends to keep the multiprocessors on the device as busy as possible. It follows that a poorly balanced workload will result in suboptimal performances. Hence, it is important to implement a correct application design with an optimal task distribution on threads and blocks. The proposed toolbox aims to spare the developers from such an effort.

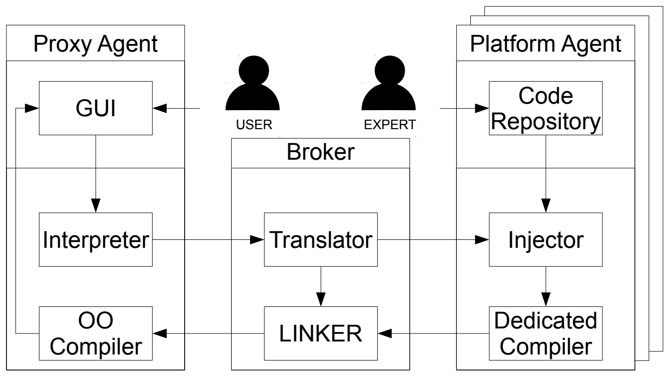

Fig. 1. An overall schema of the developed agent oriented system

III. The proposed toolbox

To strictly separate all the algorithmic development related to the application domain from different concerns (i.e. the GPU handling), which should not be taken into account by the programmer. All the GPU device-related managements are performed automatically by the proposed toolbox (e.g. the choice of the best number of threads and blocks, the needed modifications to the code in order to enable it for asynchronous stream execution, etc.).

This toolbox aims at providing a simplified and modular support for GPU computing that developers could use without having to learn how to program in CUDA. The purpose of this work is to develop such a toolbox for OO Programming to run specific tasks on the MIMD environment provided by a GPU accelerator without any need to divert from an OO paradigm and the related OO language (e.g. Java).

Figure 1 shows the proposed solution, which consists of three main agents:

- Proxy agents
- Broker
- Platform agents

The *proxy agent* provides a graphical user interface (*GUI*) which could be typically intended as a web portal to upload OO code complying with several syntactic constraints (such as the proper use of annotations). The uploaded code is then interpreted by an *interpreter* software module which creates an XML file to instruct the *translator* on the behaviour that some portions of code have to obtain. Finally, the *proxy agent* contains the OO oriented *compiler* which has the responsibility to link all the produced software modules with the unchanged portion of the original OO code and then compiles it creating an executable binary. The binary is finally returned to the user by means of the same GUI when ready.

The said *translator* module is part of the *Broker*, it receives an XML description of the behaviours from the *interpreter*. While the latter has the responsibility to detect the code behaviours and accordingly prepare an input for the translator, the interpreter has nothing to with the translation of code itself. The translator maintains a reference to the different *platform* *agents* which are designed to match different hardware infrastructures. The translator then is able to understand the interpreted behaviours, and then choose the proper architecture-oriented broker agent. Once the broker has been chosen, the translator instruct it to inject some portions of translated code, on the other hand the translator itself prepares the unchanged code to be linked with the compiled device-dedicated software modules. The *linker* module has the latter responsibility, linking the remaining OO code with the generated device-dedicated executables. The latter responsibility is all but trivial, since it is up to the linker any choice regarding the best linking approach (e.g. whether to use external functions, native interfaces, or other approaches in order to permit to let the generated binary to be called from the OO portion of the code).

The effective injection of code and generation of executable software modules is performed within the dedicated *Platform agent*. The core module of the *Platform agent* is the *injector*. As the name suggest the responsibility of this module is to inject device oriented code in order to create a separately compiled software-module to be then linked by the broker, as said before. The *injector* has knowledge of a selection of codes within a *related repository*. Basing on the indications given by the *translator*, the *injector* requires the corresponding code to the *repository*. The latter is maintained by experts of the target architecture to which the repository is related. The proposed approach is similar to a *wiki* project where experts add new code to be used. Of course the uploaded code must be compliant with all the standards of the device for which it is intended, and, moreover, should be accompanied by an adeguate descriptor within the constraints of the presented agent oriented system itself. Finally the injected code si compiled with a *dedicated compiler*, a binary file is produced and passed to the linker which performs its duties closing the cycle.

In the following section we will give some details regarding the injection procedure and the involved modules.

IV. THE DESIGNED MODULES

A. Compile time

In our approach, fragments of device-related code are automatically linked by using a predefined library of common functions of general purpose. Moreover, the designed system makes it possible to define custom CUDA compliant tasks to be executed on the GPU. At runtime, a component, which makes use of *aspect-oriented* programming, provides with the optimal management of the memory transfers and allocation by monitoring the effective allocations, initialisations, and values of the stored variables both on the host and device. By including the predefined classes or by using the precompiled executable as a linked binary, it is possible to use a predefined set of functions and also to implement custom CUDA compliant functions and then invoke them within an OO paradigm by means of Java Native Interface (JNI).

The approach provides the developer with a set of functions that, when invoked, take care of all needed management, including: the data transfers between CPU and device, the memory allocations or the use of pinned memory, the possible asynchronous execution of different threads, the optimal sizing and dimensioning with respect to best performing number of threads and blocks. Such functions are implemented as part of a set of choices made by the toolbox in order to obtain a CUDA compliant code which satisfies certain requirements that we call *behaviours*.

A behaviour is a set of fixed parameters concerning the management of the GPU card and all the related optimisation that does not involve the application logic. While the developer is responsible for the application logic, the implementation of the said *behaviour* (therefore all the choices and related implementation in terms of specific kernels, functions, parameters and strategies) are up to the presented support. Finally, the developer can also freely compose predefined behaviours or create new ones.

B. Code repository

A predefined set of behaviours and related implementing functions are included in a code repository, so that the programmer will have no need to directly implement CUDA kernels and calls, nor even to use C or C++ languages. With our approach, a developer could write his code in an OO programming language such as Java, an then use a few of the classes in our repository to provide some parameters, in order to configure the whole ensemble of application and CUDA code (examples of parameters are the number of threads, blocks, the maximum bandwidth, etc..) or let our toolbox take all the decisions.

As far as CUDA compliant code is concerned, this is implemented by the toolbox, some function pointers are predefined and enlisted so that the application developer will have the possibility to choose between a given set of functions, or manually add to the list a custom function written in CUDA C compliant code. In this way the developer could even write non-OO code (specifically CUDA code) and then use it within a more comprehensive OO application.

Any function defined in the library, hence also any custom function, is a `__device__` function: the inputs of such GPU functions consist of a unique pointer to the ensemble of data, operands and outputs; in addition to this pointer some control data are given. Note that such a code will be generated by the toolbox, which takes care of all such details. These functions are then executed on the GPU device and called by a properly generated `__global__` kernel.

The provided functions work for an arbitrary number of parameters, (i.e. operands) and functions.

C. Code injection choice

This agent oriented system was created to make use of several classes that properly realise a usable set of data assisting the computation to be performed on a GPU. These classes are adapted and interpreted as C-like defined structures of standard types, which are then transferred to the GPU device.

The JNI layer provides the needed "glue" to manage calls and data transfer towards the C++ side, which will use such classes as primitive structures. While the memory address of an object is not available under a Java framework, once objects are passed by means of JNI calls to the C++ layer, it becomes possible (within the C++ portion of code) to manipulate and pass data by means of their memory addresses. This makes it possible to ignore the number of dimensions for arrays, matrices, tensors, handled by such data types.

An important strategy has been used to reduce the size of data transferred and consists of an a priori selection and rearrangement of the operands and functions encoded, as said before, in unique arrays.

Since the application developer has to provide the starting and ending points of the operands, before any allocation or communication in and to the device, the toolbox rearranges only the necessary part of the data in a communication buffer. Under the stated conditions the memory allocated and the data transferred to the device are minimised, on the other hand the total size of such a buffer maximises the bandwidth usage. With this selection of data another advantage is to minimise the operations on the device due to the indexing of the operands.

Another important feature of the proposed toolbox is the simplified interface to memory transfer between host and device. It is known that, as far as execution time is concerned, generally the more costly part is memory allocation and data transfer from the host to the GPU device and vice versa, which, for the best part, is at the origin of the total overhead. Memory transfer is not only expensive at runtime, indeed it is considered the ticklish and misleading part in GPU programming being also expensive in terms of coding time.

This toolbox takes care of memory allocations on the device and offers an advanced management of communications between host and GPU device. For this reason, e.g. when a variable is used twice on the GPU device during the same execution of the program, and if it is not reassigned or redeclared in the meantime, the toolbox will avoid to repeat a memory transfer, preserving a copy on the device for future use. This feature gives an easy way for the programmer to develop GPU-ready code without any need to take care of these tricking side considerations, focusing only on the algorithm she wants to implement.

D. Translator module

As said above, our toolbox uses several independent agents in order to generate device oriented software modules that can be connected with any other Java application code by using the Java Native Interface. The software modules are created by means of a common nvcc compiler without any other precompiler. In fact, it comes with the needed computational libraries which can also be precompiled and linked to an existing software system.

In order to enable application developers to produce modular code, the proposed translation system makes use of behaviours to obtain certain features of the code. Such behaviours could be intended for CUDA-compliant code in the same way as *Design Patterns* are intended for OO code. In such a context, aspect-oriented programming (AOP) has been proven effective to implement OO design patterns while preserving the independence of classes and the separation of concerns [36]. In the same way, AOP is useful in order to connect an OO application with CUDA native GPU code.

Since the proposed toolbox takes advantage of some specified parameters and several annotations given by the developer in order to identify the correct behaviour (or composition of behaviours), the adopted solution could be classified as an Annotated Aspect oriented solution (AA). In such an AA solution several aspects are responsible for the interception of the relevant OO methods, whose execution is ultimately substituted with native code, interacting with the remaining portion of the software by means of JNI. In this case, we want to run JNI instances driven by some parameters. Some of them are embedded into the OO code, others should be evaluated at runtime, e.g. the core occupancy, whether or not to use pinned memory, the effective bandwidth utilisation as in equation (2). While the presented toolbox makes all the needed computation by means of a meta-layer which reflects on the OO code, in order to correctly interpret the desired behaviour it is up to the developer to annotate his application code. In order to minimise this concern we have developed an easy way to set up all the needed parameters and to select the portions of code to parallelise.

```
public @interface GPUstream {
    int value();
}
public @interface GPUparal {
    String fixed();
}

public class Behaviour {
    private static Behaviour b = new Behaviour();
    private String status;
    private String tmpStatus;

    private Behaviour() { }

    public static Behaviour getInstance() {
        return b;
    }

    public String eval(String s) { }
    public String wise() { }
    public String add(String s) { }
    public void set(String s) { }

    public void init(String a) {
        add(a);
        add(eval(tmpStatus));
        add(eval(wise()));
    }

    public String get() { }
}
```

Fig. 2. Examples of predefined annotations and classes

Some annotations are given by the toolbox itself and are part of a library. Among such default annotations some of them are used to let the aspects inject the appropriate code in the right points within the code. Figure 2 shows annotation `@GPUparal` that allows identifying a class that has to be substituted with CUDA compliant code, then some code is executed on the GPU card and connected with the OO software by means of JNI. Such a parallel execution could be organised into several streams by means of a `@GPUstream` annotation. The latter takes as input an ID in order to univocally identify an execution stream (mandatory in case of asynchronous execution and pinned memory utilisation). Moreover, `@GPUparal` annotation allows the developer to define mandatory behaviours for certain classes or methods. Such mandatory behaviours could be defined along with the implementation of application methods and classes and become proper directives, when the

```
public class Main {
    @GPU_param(threads=64, blocks=8, async=1,
        pinned=1, buff=1024, streams=1, mixbehav=0,
        fixed="threads,async,pinned")
    public static void main(String[] args) {
        // ...
    }
}

@GPUparal(fixed="default")
public class MyClass {

    @GPUstream(1)
    public void bigMethod1() {
        // ...
        Behaviour.getInstance().set("none");
        smallMethod_S1();
        // ...
    }

    public void bigMethod_S0() {
    }

    @GPUstream(2)
    public void bigMethod3() {
        // ...
        otherClass.smallMethod_OC();
        // ...
        smallMethod_S2();
        // ...
    }
}
```

Fig. 3. Examples of application customisation by using Java annotations

implemented methods (or classes) are called (or used).

Some specific behaviours can be implemented to be enabled at a given moment in the code, e.g. before a call to a method, then method `set()` on an instance of class `Behaviour` has to be called (see also Figure 3). The said method `set()` would then override any other behaviour except for the mandatory behaviour.

Figure 3 shows how an application takes advantage of the proposed toolbox by means of Java annotations. These set important parameters, or communicate some global behaviour, which is then implemented for all the parallelised code. Another behaviour could be introduced by the aspects when appropriate.

E. Injection module

The developed aspect `GPUinjector` (see Figure 4) takes into account the behaviour resulting from: (i) class-related annotation `@GPU_parallel`, (ii) method-related annotation `@GPUstream`), and (iii) the directives given by means of predefined annotation `@GPU_param`. When the aspect intercepts a called method for a given instance, it observes all behavioural directives (given on the code by means of the method `set()` for class `Behaviour`). Moreover, it takes into account the overall ensemble of parameters and circumstances that intervene at runtime. This latter reasoning could lead the aspect to find a more profitable or advantageous setup in order to enrich or modify the given set of behaviours (except in the case of mandatory behaviours).

At the beginning of an application execution, as soon as class `Behaviour` is loaded, it is populated by proper data values, then using the mandatory instructions, several behaviours are configured. Such mandatory instructions, as well as all the other initialisation directives are given in the application code by means of the default annotations.

```
public aspect GPUinjector {
    pointcut GPUpar(GPUparal ann, Object obj):
        this(obj) && execution(@GPUparal void *.*(..))
        && @annotation(ann);

    void around(GPUparal ann, Object obj):
                        GPUpar(ann, obj) {
        try {
            // ...
            Behaviour.getInstance().init(ann.fixed());
            // ...
        } catch (Exception e) { }
    }
}

extern "C" JNIEXPORT void JNICALL
Java_Injected_CUDAcode(JNIEnv* env, Gstruct gpudata){
    env-> ...
}
```

Fig. 4. Aspect for CUDA code injection

When an instance of an annotated application class will be intercepted by `GPUinjector`, the class-related annotations are taken and then stored on an appropriate hash table handled by the aspect. The same elaboration is performed for method-related annotations, which are stored in another dedicated hash table. Data stored in such hash tables are shared by the several instances of the same class, once they are intercepted, since the same parallelisation is desired for all instances of the same class.

Data gathered by method-related and class-related annotations are then merged with the mandatory behaviours and other specifications given by the application developer in order to exclude conflicting behaviours. Then, the resulting behaviours are given by the attribute `tmpStatus`, as extracted by the aspect from the annotations. Such behaviours are evaluated by method `eval()` in class `Behaviour` in order to check the compatibility with the mandatory behaviours before modifying the general behaviour encoded as a string `Status`.

Finally, aspect `GPUinjector` enables us to integrate on-the-fly new behaviours for some advantageous circumstances. This latter integration is made by means of method `wise()` in class `Behaviour`. After the evaluation the behaviours are added (or modified) with method `add()` on the same class.

When the whole image of the behaviour is composed, then the aspect calls the code generator that joins preexistent portions of code related to each behaviour composition (or custom made CUDA compliant code linked by the developer to a certain composition on default or custom behaviours). Figure 5 shows the Java code and annotations and calls that connect with our toolbox.

V. CONCLUDING REMARKS

The possibility to have an easy to use and modular toolbox for GPGPU programming opens an entire new range of possibilities in the field of fast and performance oriented computing. By means of our toolbox the developer need not use any external or proprietary compiler. Consequently, this toolbox offers virtually unlimited reusability with the possibility to link

```
import java.lang.annotation.*;
import Gclass.*;


@GPUparal("none")
public class MyGPUclass {
    public void myGPUmethod(Ptype data) {
    }
}

public class Test {
    @GPU_param(threads=48, async=0, mixbehav=1,
        fixed="threads,split1D")
    public static void main(String[] args) {
        Ptype data = new Ptype();

        Behaviour.getInstance().set("default,split1D");
        MyGPUclass.myGPUmethod(data);

    }
}
```

Fig. 5. Example of Java code before injection

with CUDA-compliant code. The implementation of advanced features is made by using aspect-oriented code. In this way, it is possible to have an high customisation level for the definition of new behaviours and performances improvements due to an advanced management of the allocation and freeing of memory on the device. This will provide means to control the lifecycle of variables stored on the device. All this without compromising the modularity and simplicity of the implementation which are the main driving forces of this work.

Moreover, the presented toolbox works as an integrated translational utility for the automatic conversion between sequential OO code as well as for the integration of CUDA-compliant code, providing an advanced interpretation method. In this way programmers that intend to make use of the advantages offered by this toolbox will be able to reuse written code by translating it into a GPU-enabled implementation, with a robust compatibility between this toolbox and any independent OO code.

References

[1] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Proceedings of International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 159–168, IEEE, 2010.

[2] C. Napoli, G. Pappalardo, and E. Tramontana, "Improving files availability for bittorrent using a diffusion model," in *23rd IEEE International WETICE Conference*, pp. 191–196, IEEE, 2014.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of Parallel and Distribuited Computing*, vol. 68, pp. 1370–1380, 2008.

[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] A. Rueda and L. Ortega, "Geometric algorithms on cuda," *Journal of Virtual Reality and Broadcasting*, no. 200, 2008.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," *ECOOP 97 Object Oriented Programming*, pp. 220–242, 1997.

[7] G. Borowik, M. Woźniak, A. Fornaia, R. Giunta, C. Napoli, G. Pappalardo, and E. Tramontana, "A software architecture assisting workflow executions on cloud resources," *International Journal of Electronics and Telecommunications*, vol. 61, no. 1, pp. 17–23, 2015. DOI: 10.1515/eletel-2015-0002.

[8] R. Giunta, G. Pappalardo, and E. Tramontana, "An aspect-generated approach for the integration of applications into grid.," in *Proceedings of the symposium on Applied computing*, ACM, 2007.

[9] C. Napoli, F. Bonanno, and G. Capizzi, "An hybrid neuro-wavelet approach for long-term prediction of solar wind," in *IAU Symposium*, no. 274, pp. 247–249, 2010.

[10] R. Giunta, G. Pappalardo, and E. Tramontana, "Superimposing roles for design patterns into application classes by means of aspects," in *Proceedings of Symposium on Applied Computing (SAC)*, ACM, 2012.

[11] M. Woźniak, Z. Marszałek, M. Gabryel, and R. K. Nowicki, "Modified merge sort algorithm for large scale data sets," *Lecture Notes in Artificial Intelligence - ICAISC'2013*, vol. 7895, pp. 612–622, 2013. DOI: 10.1007/978-3-642-38610-7_56.

[12] E. Tramontana, "Automatically characterising components with concerns and reducing tangling," in *Proceedings of Computer Software and Applications Conference (COMPSAC) workshop QUORS*, IEEE, 2013.

[13] C. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[14] C. Napoli, G. Pappalardo, and E. Tramontana, "An agent-driven semantical identifier using radial basis neural networks and reinforcement learning," in *XV Workshop "Dagli Oggetti agli Agenti"*, vol. 1260, CEUR-WS, 2014.

[15] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures.," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 89–108, ACM, 2010.

[16] A. Fonseca and B. Cabral, "Aeminiumgpu: An intelligent framework for gpu programming," in *Proceedings of Facing the Multicore-Challenge III, At Stuttgart*, 2012.

[17] M. Boyer, K. Skadron, and W. Weimer, "Automated dynamic analysis of cuda programs," in *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[18] M. Mongiovi, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana, "Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications," in *Proceedings of Symposium on Applied Computing (SAC)*, ACM, 2015.

[19] A. Calvagna and E. Tramontana, "Delivering dependable reusable components by expressing and enforcing design decisions," in *Proceedings of Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, pp. 493–498, IEEE, July 2013.

[20] F. Bonanno, G. Capizzi, S. Coco, C. Napoli, A. Laudani, and G. Lo Sciuto, "Optimal thicknesses determination in a multilayer structure to improve the spp efficiency for photovoltaic devices by an hybrid fem—cascade neural network based approach," in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2014 International Symposium on*, pp. 355–362, IEEE, 2014.

[21] R. Giunta, G. Pappalardo, and E. Tramontana, "A redundancy-based attack detection technique for java card bytecode," in *Proceedings of International WETICE Conference*, pp. 384–389, IEEE, 2014.

[22] C. Napoli, G. Pappalardo, and E. Tramontana, "Using modularity metrics to assist move method refactoring of large systems," in *Proceedings of International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pp. 529–534, IEEE, 2013.

[23] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*, IEEE, 2013.

[24] F. Bonanno, G. Capizzi, and C. Napoli, "Some remarks on the application of rnn and prnn for the charge-discharge simulation of advanced lithium-ions battery energy storage," in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2012 International Symposium on*, pp. 941–945, IEEE, 2012.

[25] E. Tramontana, "Detecting extra relationships for design patterns roles," in *Proceedings of AsianPlop*, March 2014.

[26] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in openmp," in *Artificial Intelligence and Soft Computing*, pp. 22–33, Springer International Publishing, 2014.

[27] G. Capizzi, F. Bonanno, and C. Napoli, "A new approach for lead-acid batteries modeling by local cosine," in *Power Electronics Electrical Drives Automation and Motion (SPEEDAM), 2010 International Symposium on*, pp. 1074–1079, IEEE, 2010.

[28] M. Ioki, S. Hozumi, and S. Chiba, "Writing a modular gpgpu program in java," in *Proceedings of workshop on Modularity in Systems Software (MISS)*, (New York, NY, USA), pp. 27–32, ACM, 2012.

[29] H. Nguyen, *GPU Gems 3*. Addison-Wesley, 2008.

[30] C. Napoli, G. Pappalardo, E. Tramontana, Z. Marszałek, D. Połap, and M. Woźniak, "Simplified firefly algorithm for 2d image key-points search," in *2014 IEEE Symposium on Computational Intelligence for Human-like Intelligence*, pp. 118–125, IEEE, 2014.

[31] G. Capizzi, F. Bonanno, and C. Napoli, "Recurrent neural network-based control strategy for battery energy storage in generation systems with intermittent renewable energy sources," in *IEEE international conference on clean electrical power (ICCEP)*, pp. 336–340, IEEE, 2011.

[32] M. Woźniak and D. Połap, "Basic concept of cuckoo search algorithm for 2D images processing with some research results : An idea to apply cuckoo search algorithm in 2d images key-points search," in *SIGMAP 2014 - Proceedings of the 11th International Conference on Signal Processing and Multimedia Applications, Part of ICETE 2014 - 11th International Joint Conference on e-Business and Telecommunications*, (28-30 August, Vienna, Austria), pp. 157–164, SciTePress, 2014. DOI: 10.5220/0005015801570164.

[33] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A novel cloud-distributed toolbox for optimal energy dispatch management from renewables in igss by using wrnn predictors and gpu parallel solutions," in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2014 International Symposium on*, pp. 1077–1084, IEEE, 2014.

[34] Nvidia corporation, *NVIDIA CUDA Compute Unified Device Architecture programming guide*. 2007.

[35] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappalà, "A cloud-distributed gpu architecture for pattern identification in segmented detectors big-data surveys," *The Computer Journal*, p. bxu147, 2014.

[36] R. Giunta, G. Pappalardo, and E. Tramontana, "Aspects and annotations for controlling the roles application classes play for design patterns," in *Proceedings of the Asia Pacific Software Engineering Conference (APSEC)*, pp. 306–314, IEEE, 2011.