Towards Code Generation from Design Models for Embedded Systems on Heterogeneous CPU-GPU Platforms

Federico Ciccozzi
Mälardalen University, MRTC
Västerås, Sweden
federico.ciccozzi@mdh.se

Abstract—The complexity of modern embedded systems is ever increasing and the selection of target platforms is shifting from homogeneous to more heterogeneous and powerful configurations. In our previous works, we exploited the power of model-driven techniques to deal with such complexity by enabling the automatic generation of full-fledged functional code from UML models enriched with ALF action code. Nevertheless, the scope was bounded to CPU-based platforms.

In this work we propose a preliminary definition of the means to build upon the current code generator to enable the generation of code targeting heterogeneous platforms, more specifically conceiving mixed CPU-GPU configurations. The aim is to minimise the effort of the user in modelling platform-related information by embedding the greatest feasible amount of it into the transformation process.

I. INTRODUCTION

In this paper we lay the foundations towards the automatic generation of full-fledged functional code to be run on complex heterogeneous platform configurations based on both CPUs and GPUs. Previously, we developed a model-driven technique [1] for preservation of extra-functional properties that was able to generate code to be run as singleprocess\(^1\) application on singlecore CPU-based platforms. Afterwards, we proposed and developed extensions in order to enable the generation of more complex homogeneous configurations, such as multiprocess on single and multicore, that we employed for deployment assessment in [2]. In order to widen the scope of the approach in [2], we hereby focus on extending our code generator towards the generation of code to be run on heterogeneous platforms based on a combination of CPUs and GPUs.

For achieving our purposes, we exploit the power of Model-Driven Engineering (MDE) which aims at easing the system development by exploiting models as abstractions of the real phenomenon. In fact, the use of models can reduce the complexity of the problem and focus on the concepts that matter in the design of the system by reasoning in terms of domain-specific concepts [3]. One of the major objectives of MDE is to provide automated generation of code to be executed on specific target platforms. Automating the code generation phase is a critical task that has the potential to make MDE an eligible approach to substitute code-centric approaches. Qualitative factors are also improved, such as consistency-by-construction of the generated code to the different modelling artefacts along the entire development process (i.e., models at different abstraction levels and generated code) [4] as well as overall quality and maintainability of the generated code. In fact, by producing 100% of target code from design models, consistency can be kept and thereby results from analysis performed at model level are likely to have a high degree of validity also at code level (and the other way around).

Effective code generation can positively impact economic factors, such as time-to-market as well as overall costs and risks. This can be achieved thanks to, e.g., the ability of abstracting from details not needed at design level and that are typical of code-centric approaches. Moreover, generated code aims to achieve higher and more consistent quality than hand-written code with respect to errors, maintainability and readability.

In [1] we described our automated round-trip engineering support for MDE of embedded systems and the outcomes have been validated in industrial settings where the necessity to extend the generation of code to be able to deal with more complex platform configurations arose. In this work we address extensions to enable the generation of mixed CPU-GPU applications and thereby broaden the capabilities of the deployment assessment approach presented in [2].

The remainder of the paper is organised as follows. Section II depicts the means we identified for enabling the generation of applications to be run on CPU-GPU platforms from design models. A summary of the current state of the art related to the addressed problem is given in Section III. The paper is concluded in Section IV by the current status together with the upcoming planned work.

II. HETEROGENEOUS CPU-GPU CODE GENERATION

In several domains (e.g., medical, automotive, aerospace) embedded systems are expected to process massive amounts of data, even in real-time. Aiding in fulfilling these expectations, the development of hardware technologies towards heterogeneous configurations makes embedded systems able...
to handle, e.g., very high input data rates [5]. A typical scenario would be represented by input data coming into a CPU, which in turn may exploit one or more GPUs as coprocessors for parallel processing of large blocks of data. Such a technology shift from homogeneous to heterogeneous platforms raises a number of new research issues on both the modelling and coding of embedded systems.

On the other hand, the introduction of heterogeneity gives us the possibility to enable faster computation and generally increase the performances of the generated implementation. On the other hand it adds an additional level of complexity in the platform and deployment configuration and therefore complicates the code generation process. Due to the fact that CPUs and GPUs employ different formalisms and mechanisms for code execution, as well as different programming languages, the transformation process has to be enhanced in order to be able to map model entities to code artefacts written in different target languages. Moreover, the transformation will have to generate the communication code needed for the interaction between CPUs and GPUs. Finally, in order for the code generation to account these different levels of granularity, fine-grained deployment information has to be accurately modelled at design level.

In this section we unwind the means we identified for enabling the generation of applications to be run on CPU-GPU platforms from design models.

A. Context

In order to build a code generator aiming at producing full-fledged code from models, syntax and semantics of the employed modelling formalisms should be somewhat constrained. Especially in the case of very expressive and generic languages, such as UML, suitable syntax and semantics restrictions should be defined (e.g., through UML profiling) in order to delimit the code generation’s scope. For this reason, we define the design models by means of the CHESS-ML[6], which is a UML-profile including a subset of the MARTE profile, and modelled following the component-based design pattern described in the UML by which the specification of a system is defined as an assembly of components communicating via required and provided interfaces (exposed by ports). Moreover, the behaviour specification is modelled in terms of UML state-machine diagrams enriched with Action Language for Foundational UML (ALF) code describing each operation’s behaviour (for more details please refer to [1]). In fact, modelling behaviours by means of target languages (e.g., C/C++, Java) instead of model-aware formalisms (i.e., ALF) would outright jeopardize the consistency at modelling level and therefore between models and generated code due to the abstraction gap between modelling formalisms and such languages.

The target languages are C++, for code portions running on CPU, and CUDA C/C++ [7] for code portions to be deployed on GPU. The focus here is to automate the generation of C++ together with CUDA C/C++ and the needed CPU-GPU communication code depending on how the modelling elements (e.g., components and functions) are deployed on the different processors. For simplicity reasons we will focus on a single CPU while considering $N \geq 1$ GPUs.

B. Modelling Deployment

In order to be able to automate the generation of code from models to target complex platforms, proper modelling means have to be defined for those points of variability that cannot be embedded in the transformation process. First of all, assuming that, by default, components’ functions are allocated to CPUs, the modeller should be able to define the allocation of functions to GPU devices. Fig. 1 depicts a simple example of modelling deployment on a heterogeneous configuration. More specifically, the system is composed by components $A$ and $B$ which are allocated to a CPU, and $A$’s function $f()$ is deployed on GPU with $ID = 1$. Since the other $A$’s function, $k()$, as well as $B$’s call $A()$ are not specifically allocated to any GPU, they will be deployed on the CPU. Moreover, grid dimensions ($gridD = x$) and thread block ($blockN = y$), that will be used to call the kernel function implementing $f()$, are defined as a decoration of the allocation.

![Figure 1. Modelling Heterogeneous Deployment](image)

C. Generating the Application

When it comes to the generation of the application, we need to enhance the generation process in order to generate (i) the needed communication code to call $f()$, and (ii) the actual code specified for $f()$ in the model in terms of ALF to a CUDA C/C++ function (or set of functions). Let us suppose that $B$’s function call $A()$ calls both $A$’s $f()$ and $k()$ as depicted in ALF-like format in Code 1.
As depicted in the code, function \( f() \) computes the (parallel) sum of the input parameters \((\text{in 1}, \text{in 2})\) and put the result in the output parameter \( \text{out result} \). While \( k() \) and \( call_A() \) are generated as standard C++ functions, and therefore can be handled by the current code generator, \( f() \), which is deployed on a GPU, needs to be translated into CUDA C/C++ and communication code has to be generated in order to allow \( call_A() \) to call it. Since the focus of this work is on generating such communication code, we suppose that the body of \( f() \) is already written in a CUDA-like fashion. In order to be able to do this, specific CUDA built-in variables (e.g., threadIdx, blockIdx, blockDim) are modelled as data-types in the UML/ALF model and therefore can be understood by the ALF language as the rest of the surrounding model. Doing so, model-awareness of the action language is not jeopardized and consistency-by-construction can still be ensured.

The idea is, therefore, to enhance the generation process to produce communication code whenever the code generator runs into a call \( call_A() \) to a function \( f() \) allocated to a GPU. Code 2 depicts the generated C++ \( call_A() \) function as well as the generated CUDA code in terms of \( f() \) function, representing the translated body of \( f() \), and \( f\_caller() \), which represents the function implementing the communication code needed to call the actual computation \( f() \).

```
1 // call_A in ALF-like
2 public call_A(){
3     B.k();
4     int opl = 5, op2 = 10, res = 0;
5     B.f_caller(op1,op2,res);
6 }
7
8 // f() in ALF-like
9 public f(in o1, in o2, out result){
10     int i = threadIdx.x +
11         (blockIdx.x * blockDim.x);
12     result[i] = o1[i] + o2[i];
13 }
```

As depicted in the code snippet, function \( f() \) computes the (parallel) sum of the input parameters \((\text{in 1}, \text{in 2})\) and put the result in the output parameter \( \text{out result} \). While \( k() \) and \( call_A() \) are generated as standard C++ functions, and therefore can be handled by the current code generator, \( f() \), which is deployed on a GPU, needs to be translated into CUDA C/C++ and communication code has to be generated in order to allow \( call_A() \) to call it. Since the focus of this work is on generating such communication code, we suppose that the body of \( f() \) is already written in a CUDA-like fashion. In order to be able to do this, specific CUDA built-in variables (e.g., threadIdx, blockIdx, blockDim) are modelled as data-types in the UML/ALF model and therefore can be understood by the ALF language as the rest of the surrounding model. Doing so, model-awareness of the action language is not jeopardized and consistency-by-construction can still be ensured.

The idea is, therefore, to enhance the generation process to produce communication code whenever the code generator runs into a call \( call_A() \) to a function \( f() \) allocated to a GPU. Code 2 depicts the generated C++ \( call_A() \) function as well as the generated CUDA code in terms of \( f() \) function, representing the translated body of \( f() \), and \( f\_caller() \), which represents the function implementing the communication code needed to call the actual computation \( f() \).

```
1 // call_A in C++ (.cpp file)
2 void call_A(){
3     B.k();
4     int opl = 5, op2 = 10, res = 0;
5     B.f_caller(op1,op2);
6 }
7
8 // .._cpp file ______
9 // call_A in C++ (.cpp file)
10 void call_A(){
11     B.k();
12     int opl = 5, op2 = 10, res = 0;
13     B.f_caller(op1,op2);
14 }
```

Practically, the following steps are performed to generate the functions' code:

- **Step 1** - Create a C++ function related to \( call_A() \) (lines 1-7).
- **Step 2** - Create a CUDA C/C++ function \( f() \) function (lines 10-18) which will reflect the parallel computation modelled as \( f() \) in Fig. 1.
- **Step 3** - Create a CUDA C/C++ function \( f\_caller() \) (lines 20-47) which will be called by \( call_A() \) and that performs all the CUDA-related operations needed to call \( f() \).

More specifically regarding Step 3, a pointer for each of the parameters (both \text{in} and \text{out}) of \( f() \) is declared and given memory through the \text{cudaMalloc()} API (lines 23-29). At this point, the pointers are made to point to the values carried by the parameters by copying host memory to device memory through the \text{cudaMemcpyToSymbol()} API (lines 31-32). In Fig. 1 we showed how the number of grid dimensions \( (\text{gridD} = x) \) and the thread block \( (\text{blockN} = y) \) are modelled in the allocation of function \( f() \) on GPU with \( \text{ID} = 1 \). This information is now used by the code generator to declare the actual dimensions of both grid and block (lines 34-35), and the GPU's ID is used to assign the device to be used, through the \text{cudaSetDevice(ID)} API (line 37). Now we call finally call the kernel \( f() \) using the CUDA-specific syntax (lines 39-40). When the computation is completed, we copy the result, hold in the device memory, back to the host memory through the \text{cudaMemcpyFromSymbol()} API (line 42). The last step is to release the allocated resources through the \text{cudaFree()} API (lines 44-46).

The actual development of the proposed approach will be carried out by (i) defining means at modelling level to be able to model what is needed to achieve mixed
CPU-GPU code generation, (ii) improving the intermediate artefacts currently employed by the code generator to host CUDA-related information, (iii) defining model-to-model and model-to-text transformations carrying out the actual code generation.

As it can be noticed in the proposed simple example, in order to call $f$ \texttt{function}() from \texttt{call\_A}() (less than 20 code lines together), we would have needed to manually code more than 25 lines of communication code per call. This gives already a hint on the usefulness of automating the generation of communication code and therefore relieving the end-user of an error-prone and time consuming burden.

### III. Related Work

Several different approaches aiming at achieving code generation for embedded systems can be found in the literature. Despite the numerous attempts proposed, this still represents an open research issue especially when it comes to the generation of code to be run on heterogeneous platforms. The most concrete attempt to heterogeneous code generation for CPU-GPU configurations has been proposed by Rodrigues et al. in [8] where the authors define a code generation process from UML+MARTE models to OpenCL. While similar to ours in terms of underneath idea, the approach proposed in this work assumes a more detailed modelled platform-related information as well as targets mainly the GPU-related code. In the approach we propose, the aim is to provide an environment that allows end-users to freely model systems and allocate components’ functions to either CPUs or GPUs leaving the burden of communication code between CPU and GPU to the code generation process. Nevertheless, we will investigate the modelling means they propose to model GPU-related information with UML+MARTE to understand whether partial reuse would be feasible. In [9] the authors try to automate the task of determining the appropriate memory usage as well as the coding of data transfer between memories. This work could be investigated in the next steps of our research work towards possible optimizations of the code to be generated. Additionally, attempts to automatically generate C from CUDA have also been proposed, as in [10], [11], and could represent an interesting subject for comparison once our transformation of function bodies from ALF to CUDA C/C++ will be in place.

### IV. Outlook

Currently we are investigating the modelling language (UML and its profiles) to identify the most user-friendly solution that would allow the end-user to model the information needed for generating heterogeneous CPU-GPU embedded applications. Besides the structural specification of the system, focus is on the behavioural definitions in order to model mathematical C/C++ APIs. This will enable the possibility to model, in terms of action code, a wider set of computations.

As aforementioned, in this work we suppose that the actual parallel computation code is given, yet in terms of ALF as the rest of the functions’ body, by the end-user (i.e., $f()$ function in Code 1), while we focus on the generation of communication code (i.e., $f$ \texttt{CUDA}() function in Code 2). Nevertheless, the next step will be to avoid any CUDA-specific action code at modelling level, but rather generate parallel computation only if the related function is allocated to a GPU. In this way, we could exploit the deployment assessment approach proposed in [2] for an aware allocation (even through semi-automatic mechanisms) of components and functions to either CPUs or GPUs in order to, e.g., optimize performance and/or to decrease the communication overhead. The actual development of the proposed approach has already started with the improvement of the intermediate artefacts currently employed by the code generator to host CUDA-related information as well as the definition of proper model-to-model and model-to-text transformations for generating CUDA code.

### V. Acknowledgements

This research work has been carried out within the RALF3 project (Swedish Foundation for Strategic Research, http://www.mrtc.mdh.se/projects/ralf3/).

### REFERENCES


