The Journal of Systems and Software 161 (2020) 110488

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Component-based development of embedded systems with GPUs

Gabriel Campeanu[a,b], Jan Carlson[a,*], Séverine Sentilles[a]

[a] School of Innovation, Design and Engineering, Mälardalen University, Sweden
[b] Bombardier Transportation, Sweden

ARTICLE INFO

Article history:
Received 26 April 2019
Revised 19 October 2019
Accepted 2 December 2019
Available online 5 December 2019

Keywords:
Component-based development
CBD
Software components
Embedded systems
Graphics processing units
GPU

ABSTRACT

One pressing challenge of many modern embedded systems is to successfully deal with the considerable amount of data that originates from the interaction with the environment. A recent solution comes from the use of GPUs, providing a significantly improved performance for data-parallel applications. Another trend in the embedded systems domain is component-based development. However, existing component-based approaches lack specific support to develop embedded systems with GPUs. As a result, components with GPU capability need to encapsulate all the required GPU information, leading to component specialization to specific platforms, hence drastically impeding component reusability.

To facilitate component-based development of embedded systems with GPUs, we introduce the concept of flexible components. This increases the design flexibility by allowing the system developer to decide component allocation (i.e., either the CPU or GPU) at a later stage of the system development, with no change to the component implementation. Furthermore, we provide means to automatically generate code for adapting flexible components corresponding to their hardware placement, as well as code for component communication. Through the introduced support, components with GPU capability are platform-independent, and can be executed, without manual adjustment, on a large variety of hardware (i.e., platforms with different GPU characteristics).

1. Introduction

In practically all domains, ranging from vehicles and industrial systems to medical devices and home electronics, modern products rely heavily on software to provide increasingly complex functionality. Such *embedded systems*, where software is used to sense the environment of the system, determine appropriate responses, and then control the environment accordingly, constitute a very important subset of computer usage. In fact, 98% of all processors that are produced worldwide, are used in embedded systems (Helmerich et al., 2005).

Many modern embedded systems deal with huge amounts of information required to correctly understand the surroundings. Moreover, this data often needs to be processed in a very short time in order for the system to react correctly and to changes in the environment in a timely manner. To manage this, one trend in embedded systems is the use of boards with Graphics Processing Units (GPUs), providing much higher performance compared to a CPU for data-parallel applications where thousands of computation threads execute the same instructions but on different data.

Another trend in embedded systems is the usage of component-based development (CBD) (Crnkovic and Larsson, 2002). This software engineering paradigm promotes the construction of systems through composition of already existing software units called *software components*. The advantages that come with the usage of CBD include an increased productivity, efficiency and a shorter time-to-market. CBD proved to be a successful solution in development of industrial embedded systems, through the usage of component models such as AUTOSAR (AUTOSAR, 2019), IEC 61131 (John and Tiegelkamp, 2010) or Rubus (Hänninen et al., 2008).

1.1. Problem statement

Existing component models used in the development of embedded applications offer no specific GPU support, which introduces several disadvantages and diminishes the benefits of CBD when developing embedded systems with GPUs. With no GPU support in the component model, the component developer must explicitly construct components with functionality to be executed by specific processing units, i.e., either the CPU or GPU. Thus, the system developer, when constructing the component-based

* Corresponding author.
E-mail addresses: gabriel.campeanu@rail.bombardier.com (G. Campeanu), jan.carlson@mdh.se (J. Carlson), severine.sentilles@mdh.se (S. Sentilles).

https://doi.org/10.1016/j.jss.2019.110488
0164-1212/© 2019 Elsevier Inc. All rights reserved.

application, is restricted to use certain components matching the platform characteristics.

Furthermore, when developing a component with GPU capability, the component developer needs to encapsulate inside the component, specific GPU-related information required by the component to be successfully executed on the GPU. This information, explicitly addressing the characteristics of the GPU platform onto which the component will be executed, leads to the component to become specific to a particular hardware. As a result, the component has a reduced reusability between different (GPU) hardware contexts. Moreover, hard-coding inside components GPU information breaks the separation-of-concern which is a key CBD principle. For instance, the component developer would hard-code the number of GPU threads to be used by the component, making assumption about both the characteristics of the platform that will execute the component and the overall system architecture and the GPU utilization by other components.

Finally, encapsulating GPU functionality in individual components to make them transparently usable at system level, includes hard-coding aspects such as memory management. In addition to being time-consuming and error prone, this reduces the possibilities to make optimizations spanning multiple components.

1.2. Contributions

This work introduces a number of novel concepts to facilitate component-based development of embedded systems that use CPU-GPU hardware platforms. The contribution focuses on pipe-and-filter component models, and introduces the concepts of *flexible components, component groups* and *adapters*.

A *flexible component* is a light-weight component with a functionality that can be executed on either the CPU or GPU. Basically, a flexible component is a platform-agnostic component with an increased reusability, that can be executed, without any change, either on a CPU or on one of the different platforms that incorporate GPUs. The specific GPU-settings, such as the number of used GPU threads, are set through the configuration interface of the flexible components. In this way, we lift decisions that may bind components to specific contexts from component development to system level.

Through *component groups*, we provide a way to optimize groups of connected flexible components. Flexible components that are connected and are executed by the same processing unit (i.e., either the CPU or GPU) are enclosed in a component group that conceptually behaves like a single component. The component group inherits all the configuration interfaces and specific (input and output) data ports from the components contained in the group. This concept helps to improve system characteristics such as memory usage.

Due to the different characteristics of embedded platforms with GPUs, components with GPU capability require different activities (corresponding to the platform characteristics) for data communication. We improve the component communication via special artifacts called *adapters*. Depending on the platform characteristics, and the way in which components are connected, the appropriate adapters are automatically generated to achieve efficient communication.

To concretize these general concepts, we have defined their realization in the context of the Rubus component model (Hänninen et al., 2008), extending the set of standard Rubus elements with a representation of flexible components. Moreover, algorithms are defined for identifying adapter placements and grouping of flexible components into component groups, as well as a series of transformations and glue code generation that turns adapters, flexible components and component groups into regular Rubus components, to be further handled by the standard Rubus realization mechanisms.

Preliminary versions of these concepts have been described in previous publications. The initial idea of GPU components and the first Rubus extension relied on manually developed components and did not support transparent allocation to different GPU types (Campeanu et al., 2016). The notion of GPU-agnostic components was first proposed through code templates and a high-level API common to several GPUs (Campeanu et al., 2017a), and later evolved into the concept of flexible components targeting both CPU execution and different types of GPUs through code generation instead (Campeanu et al., 2017b). We have also investigated the possibility of grouping flexible components and realizing them as a single component for improved performance (Campeanu et al., 2018).

This article presents a consolidated combination of the different ideas into a coherent approach, including a reformulation of the initial adapter concept to fit the new notion of flexible components, and new rules for introducing and connecting the adapters. It also includes a number of new experiments conducted to evaluate aspects of the proposed approach. An extended version of this paper, together with a systematic literature review on the topic of GPU support in component-based systems and work on automatic optimization of GPU component allocation, can also be found in Gabriel Campeanu's PhD thesis (Campeanu, 2018).

1.3. Organization

The reminder of the paper is divided as follows. Details on the context of the work are presented in Section 2, followed by the problem description in Section 3 and the development process overview in Section 4. The solution of our work introduces flexible components (Section 5), component groups (Section 6) and adapters for efficient component communication (Section 7). The last part of the paper evaluates our solution (Section 8), presents the related work (Section 9) and the conclusions (Section 10).

2. Background

Our goal is to facilitate component-based development of embedded systems with GPUs. In this section, we introduce background information about the context, i.e., embedded systems, the component-based development methodology and GPUs.

2.1. Embedded systems and component-based development

Nowadays, computer systems are part of a majority of all developed electronic products. Compared to other types of software, the software in such *embedded systems* is typically more specialized and is subject to resource limitations in terms of memory and processing power, and a need for low power consumption. Moreover, since they interact directly with the environment through actuators (e.g. motors), they are subject to real-time constraints are often safety-critical since incorrect behaviour could result in danger to humans or the environment.

Software applications have greatly increased in size and complexity over the last decades (Liggesmeyer and Trapp, 2009), and component-based development (CBD) is one approach to tackle the challenge of efficient software development by composing software from standardized blocks called *software components*. The benefits of CBD also include easier reuse of components developed either in-house or by third-parties, further improving the development efficiency.

CBD has been successfully used in building complex desktop applications through general-purpose component models such as CORBA (OMG CORBA Component Model, 2019), .NET (Lowy, 2005),

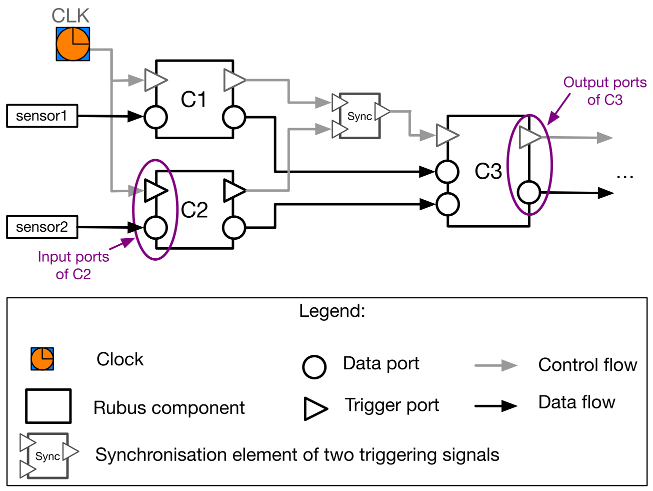


Fig. 1. Connected Rubus components.

COM (Box, 1997) and JavaBeans (Sun Microsystems, 2019). When it comes to embedded systems, the general-purpose component models lack means to handle the specifics of this domain such as real-time properties and low resource utilization (Crnkovic et al., 2011). Consequently, a number of component models specifically targeting embedded systems have been proposed.

Our work focuses on component models that utilize a *pipe-and-filter* interaction style, where components that process data behave as *filters* while the connections between components are seen as *pipes* that transfer data from one component to another. This interaction style is common in embedded systems since the simple and static control flow provides a high predictability level with respect to analysis of the temporal behavior, required to satisfy the real-time requirements of an embedded system. Among the component models that follow the pipe-and-filter style we mention ProCom (Bures et al., 2008), COMDES II (Ke et al., 2007) from academia and industrial component models such as IEC 61131 (John and Tiegelkamp, 2010) and Rubus (Hänninen et al., 2008). These component models may be applied to various embedded system areas, such as automotive (Rubus) and industrial programmable controllers (IEC 61131). Since we use the Rubus component model to concretize the proposed approach, the following paragraphs give a short introduction to the Rubus components and the component communication mechanism.

The Rubus component model follows the pipe-and-filter interaction style, with explicit separation of data and control flow. Every Rubus component is equipped with two types of ports, i.e., data and trigger ports. Through the trigger ports, the control is passed between components; similarly, data is passed using the data ports. A Rubus component is equipped with a single input trigger port and a single output trigger port; regarding data ports, a component may have one or several (input and output) ports.

Fig. 1 presents a Rubus (sub-)system composed of three connected components, i.e., *C*1, *C*2 and *C*3. At a periodic interval of time specified by the clock element *CLK*, component *C*1 is triggered through its trigger input port, i.e., it receives the control to execute its behavior. The execution semantic of the Rubus component is Read-Execute-Write. It means that *C*1 was in an inactive mode before being triggered by the clock element. Once activated, the component switches to Read mode where it reads the data from its input data port, received from *sensor1*. During Execute mode, the component performs its functionality using the input data. After the execution completion, the result is written in the output data port during Write mode, and the output trigger port is activated. The control is passed to the next connected component (i.e., *C*3) through the output trigger port, and *C*1 returns to the inactive state.

2.2. Graphics processing units

When GPUs appeared in the late 90s, they were only intended for graphics-based applications, excelling in rendering high-definition graphics scenes. Over time, however, GPUs were equipped with more general computation capabilities and became easier to program, and thus were used also in many non-graphical computationally demanding applications, such as cryptography applications (Manavski, 2007) and Monte Carlo simulations (Preis et al., 2009).

Various vendors such as Intel, AMD, NVIDIA, Altera, IBM, Samsung and Xilinx develop embedded board platforms with GPUs. The GPU is made part of these platforms in two ways, either as a discrete unit or integrated into the platform. When the GPU is discrete (referred to as *dGPU*), it has its own private memory. When the GPU is integrated (known as *iGPU*) on the same chip with the CPU, the memory is shared between the CPU and GPU. Embedded boards with iGPU architectures is the predominant platform type used in industry due to their lower cost, size and energy usage. On the other hand, dGPUs, with larger physical size and increased GPU resources, are used by systems that require higher performance.

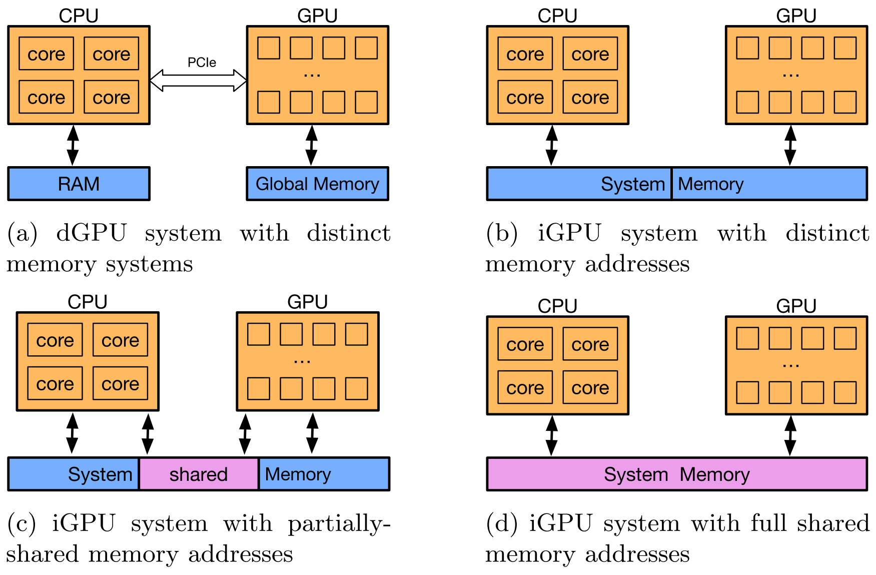
For the iGPU-based platforms, we distinguish three types of architectures regarding the memory system, i.e., distinct, partially-shared and full shared memory system. Although the CPU and GPU share the same chip, there are platforms where each processing unit has its own memory address. Other platforms, that are more advanced, have a partially-shared memory system, where a part of the memory is directly accessed by both of the processing units. The latest platforms provide a full shared memory system which can be directly accessed by the CPU and GPU.

Fig. 2 illustrates the architectures of different platforms with GPUs. Systems with dGPUs (Fig. 2(a)) are characterized by distinct memory systems, where data needs to be transfered from one system to the other via e.g., a PCIexpress bus. Most platforms with iGPUs have the same physical memories divided into distinct parts, i.e., one for the CPU and the other for the GPU (Fig. 2(b)). In this case, there is still need for data transfer activities, although with a lower transfer overhead due to the physical location of the data (i.e., on the same memory chip). There are also platforms with an optimized memory access which offer a shared virtual memory (SVM) space (Fig. 2(c)). To place data on the SVM, specific transfer activities are used; on the other hand, no specific activities are used by either the CPU or GPU to access the data on the SVM. The latest and most technological advanced architecture (Fig. 2(d)) offers simultaneous access to the same memory for both CPU and GPU, without any need for data transfer.

3. Challenges of using GPUs in component-based development

Existing component models targeting embedded systems provide no specific support for developing applications with GPU capability. Using these approaches, one way to construct component-based embedded systems with GPUs is to encapsulate, inside the component, all the GPU-specific information and operations to address the hardware platform. Encapsulating GPU information inside components brings the following disadvantages: *(i)* inefficient component communication; *(ii)* reduced component reusability and maintainability; *(iii)* reduced system design flexibility; and *(iv)* complex, error-prone and time-consuming component development.

To explain these issues in more detail, we first introduce a running example, that will be used both to elaborate the challenges and to later on illustrate our proposed solution.



(a) dGPU system with distinct memory systems

(b) iGPU system with distinct memory addresses

(c) iGPU system with partially-shared memory addresses

(d) iGPU system with full shared memory addresses

Fig. 2. Embedded platforms with different GPU architectures.

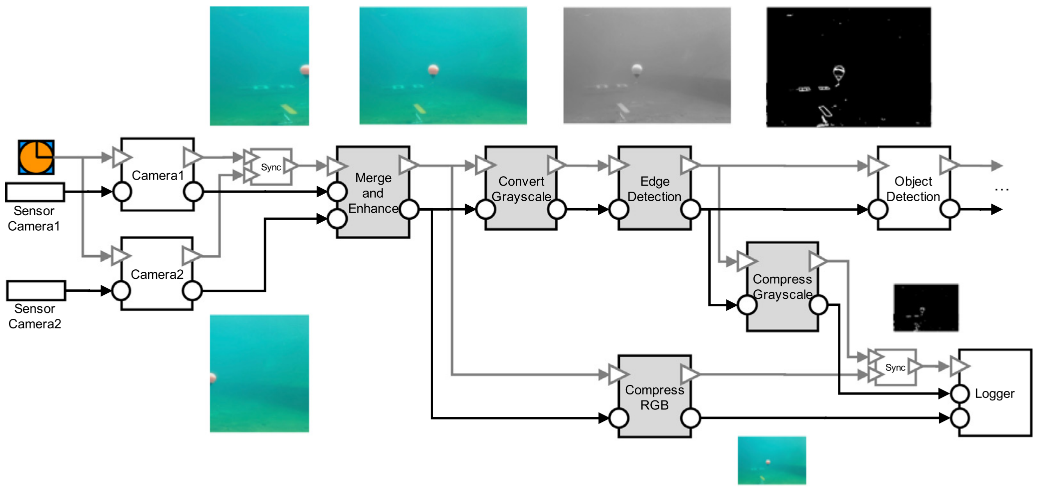


Fig. 3. The component-based vision system of the underwater robot.

3.1. Running example

The example used throughout the paper is an existing underwater robot that autonomously navigates under water, and, based of its vision system, fulfills various missions. An addition to other sensors, it relies on a system of cameras that provides a continuous flow of data regarding the environment. The embedded board contains a CPU and GPU that are integrated on the same physical chip.

Fig. 3 describes the vision system of the underwater robot, constructed using the Rubus component model. The system contains nine components, as follows: The *Camera1* and *Camera2* components receive raw data from the camera sensors and convert them to readable frames. The resulting frames are merged by the *MergeAndEnhance* component. The component also removes the noise of the merged frame, and forwards its result to the *ConvertGrayscale* component that filters it into grayscale format, from which *EdgeDetection* computes a black-and-white frame, where the white lines delimits the objects found in the frame. Based on the delimited objects, the *DetectObject* component takes appropriate actions.

Another part of the vision system is comprised of three components, illustrated in the lower part of Fig. 3. Two of the components (i.e., *CompressRGB* and *CompressGrayscale*) compress the frames of the system, while the *Logger* component, that has the purpose to record the underwater journey of the robot, registers the compressed frames.

The vision system contains five components, marked in gray, that may be executed on the GPU, due to their functionality, i.e., image processing. By executing these component on the GPU, the system may improve one or several quality properties, such as timely reactions to events in the environment.

3.2. Detailed challenges

Using the underwater robot example, we present more details on the existing challenges of component-based development of embedded systems with GPUs. Fig. 4 describes the vision system executed on a platform with distinct memory addresses for the CPU and GPU.

In order for the components to be fully compatible with any other Rubus component, each component with GPU capability needs to encapsulate all the GPU-specific information and operations, such as data transfer operations and the GPU computation settings. Both *Camera1* and *Camera2* components access the main RAM system to acquire the raw image frames, process them on the

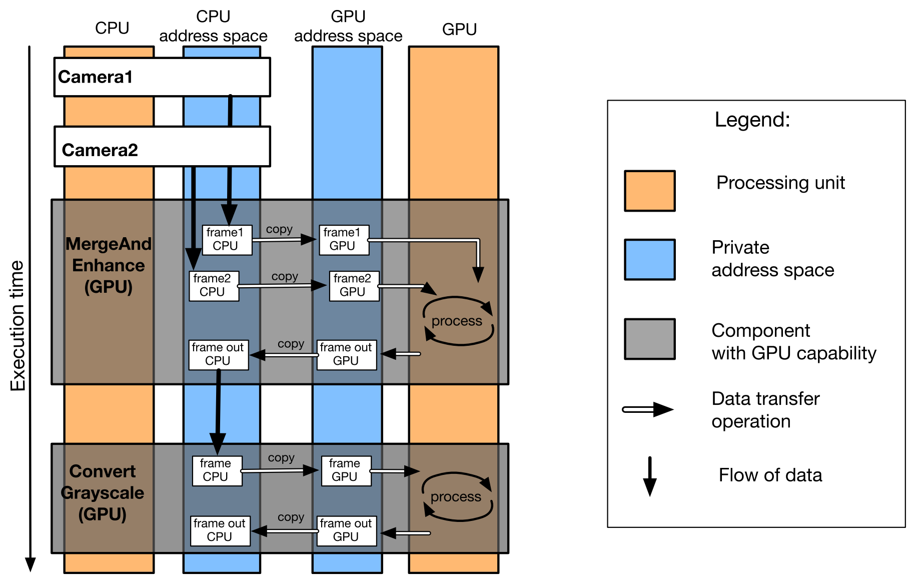


Fig. 4. Platform-related activities of the vision system.

CPU and store them back onto the RAM. Having GPU computation, the hardware activities of the *MergeAndEnhance* component are different. In addition to the RAM access to obtain the camera frames, the component needs to copy them onto the GPU memory system in order to process them. Once the frames are merged and processed, the component copies the result back onto the RAM system.

Moreover, the component developer needs to encapsulate inside the components: *i)* environment information to access the GPU, and *ii)* settings regarding the GPU threads and memory usage. These settings, referring for example to the number of GPU threads required by the component or the organization of the GPU memory needed, are linked to the physical limitations of the platform resources. Hard-coding these settings inside the component leads to a decreased maintainability of the component. Furthermore, it also influences the reusability aspect, the component being only reusable on platforms that, in best cases, have enough resources to execute it, or, in worst cases, happen to be the exact same type of platform.

The component developer, when constructing a component with GPU capability, needs to address, besides the component functionality, all the required environment information and GPU settings. This fact leads to a complex component development process, error-prone and time-consuming, and it also breaks the CBD separation of concerns principle between component and system development.

4. Overview of the proposed development process

The novelties of this work are introduced using the overview of the development process depicted in Fig. 5. Basically, the development process is composed of: *(i)* the component development activity, *(ii)* the system development activity, *(iii)* the optimization realization activity, and *(iv)* the source-code generation activity. In the component development activity, the component developer constructs the components, which are used during the system development activity to build the application architecture. During the optimization realization activity, the application is optimized and realized into a Rubus-like application (i.e., contains only Rubus constructs), while in the last activity, the application is transformed into source-code. In the following paragraphs, the development process is described in more details.

During the component development activity, the component developer constructs components of two types, i.e., *regular* and *flexible components*. While a regular component functionality is always executed by the CPU, a flexible component functionality is executed by either the CPU or GPU.

When constructing a *regular component*, the developer defines the component data ports (i.e., the number and data type of input and output ports), and writes the component behavior, i.e., functionality code constructed in a sequential manner. For constructing a *flexible component*, the developer also defines the component data ports and writes the component behavior. The flexible component functionality should be constructed in a parallel manner by using e.g., the OpenCL environment (The OpenCL specification version 2.1, 2019), in order to be executed, if decided, by the GPU. The flexible component is equipped, besides the defined data ports, with a configuration interface. This interface, transparent for the developer, will be automatically generated for each constructed flexible component, during the system generation stage.

After constructing the components, the focus of the development process is shifted on the system development activity. The design of the application is executed during the software design step, by composing the already constructed (regular and flexible) components. The output of the software design step is a component-based architecture of the desired software application. Furthermore, the platform is also modeled at this step, that is, the CPU and GPU characteristics are described (e.g., available memory). The following activity is the allocation and settings definition. The allocation may be completed either manually or semi-automatically. For the manual allocation, the system developer decides the allocation of the flexible components, i.e., on the CPU or GPU. The way that flexible components are allocated to hardware may improve one or several application properties such as performance. The output of this activity is an allocation scheme, where each component is mapped to a single processing unit. Furthermore, the system developer provides the appropriate configuration settings for each flexible component, according to its decided allocation. These settings are established using the platform model, i.e., the physical properties of the platform such as the available number of GPU threads.

In the following activity, i.e., the optimization and realization activity, connected flexible components that share the same allocation unit are grouped together in a conceptual component

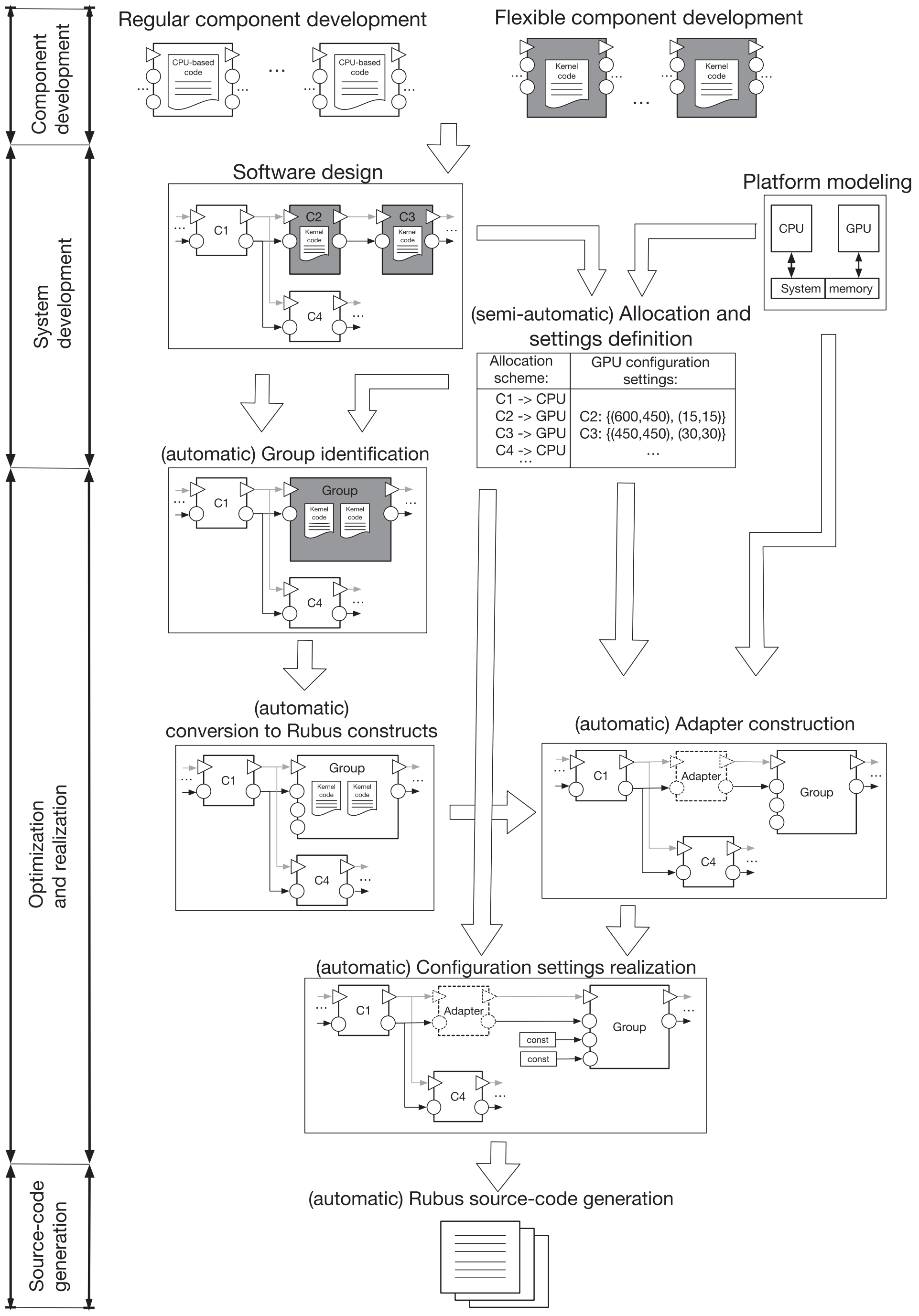

Component development
System development
Optimization and realization
Source-code generation
Regular component development
CPU-based code
Flexible component development
Kernel code
Software design
C1
C2
C3
C4
Platform modeling
CPU
GPU
System memory
(semi-automatic) Allocation and settings definition
Allocation scheme:
GPU configuration settings:
C1 -> CPU
C2 -> GPU
C3 -> GPU
C4 -> CPU
C2: {(600,450), (15,15)}
C3: {(450,450), (30,30)}
(automatic) Group identification
Group
(automatic) conversion to Rubus constructs
(automatic) Adapter construction
Adapter
(automatic) Configuration settings realization
const
(automatic) Rubus source-code generation

Fig. 5. The development process using our solutions.

referred as a *component group*. The output is an application seen as a composition of regular components and component representing component groups. During the following step, that is, the conversion to Rubus constructs step, the identified component groups are converted into regular Rubus components resulting in an application architecture that is composed of only regular components that have different allocations. In order to facilitate the communication between components allocated to different processing units, *adapter* artifacts are automatically generated during the adapter construction step. The adapter artifact connects connected components that are allocated on different processing units (i.e., using the allocation scheme) for particular platforms (i.e., based on the platform modeling). This communication artifact is transparent and generated in an automatic manner. Another step related to the realization activity is the configuration settings realization. The output of the adapter construction step is enriched with the configuration setting values provided by the system developer during the allocation and configuration activity.

The following activity in the system development is the source-code generation. The output of the configuration settings realization step is transformed during the Rubus source-code generation activity, into source-code. The code, spread in different (headers and source) files, represents the application that is ready to be executed on the platform. We mention that all introduced concepts were converted into regular Rubus constructs. For example, each component group is transformed into a regular Rubus component. In this way, the resulted system can be handled by the existing Rubus code generation.

5. Flexible component

We consider in this work that a system is designed by composing regular and flexible components. While a regular component is always executed on the CPU, a flexible component may be executed by either the CPU or GPU. Formally, we define a system to be sets of regular C_R and flexible components C_F, and their connections *Connect*:

$$\begin{aligned} S &= \langle C_R, C_F, Connect\rangle, \text{ where} \\ C_R &= \{C_{reg_1}, C_{reg_2}, \ldots, \}, \\ C_F &= \{C_{flex_1}, C_{flex_2}, \ldots, \}, \\ Connect &= \{\langle p_{from_1}, p_{to_1}\rangle, \langle p_{from_2}, p_{to_2}\rangle, \ldots\}. \end{aligned}$$

We define a component, either regular or flexible, as a tuple that contains the functionality F and the interface I consisting of ports through which the functionality can be accessed.

$$C = \langle F, I\rangle, \textit{ with } I = \{p_1, p_2, \ldots\}.$$

In the case of a flexible component, the functionality is basically the kernel code constructed using the OpenCL syntax. Written in a data-parallel manner, it can be executed by either CPU or GPU. For regular components, the functionality is a single C function, called when the component is triggered for execution.

Through the ports of the interface, a component communicates with other components, i.e., by providing (through output ports) and requiring (through input ports) data. Each port has a unique name and a specific data type that characterizes the (provided or required) data. For a port p, we let *p.name* denote the name and *p.type* the data type of p.

Due to the characteristics of the GPU, large data (e.g., images) need to reside on the GPU memory (e.g., for platforms with distinct memory systems) in order to be processed by the GPU. Consequently, ports transferring such data need to be handled differently from ports with simple data types such as *integer* or *double*, during code generation and optimization. Simple data types can be sent to the GPU as arguments of the kernel code, but for complex data we first need to ensure that the data resides in the right memory, and then a reference to the data can be sent as a kernel code argument. To support this, we must distinguish between these categories of ports at the component interface level, by introducing the predicate *p.ref* indicating if p is a reference port or not.

The ports of the interface I are grouped into two subsets, i.e., input and output. Moreover, each subset is further divided according being simple or a reference). We mention also that a (regular or flexible) component contains one single input trigger port and one single output trigger port. Therefore:

$$I = I_{in} \cup I_{out}, \text{ where}$$

$$\begin{aligned} I_{in} &= I_{sim\_in} \cup I_{ref\_in} \cup \{p_{trig\_in}\} \text{ and} \\ I_{out} &= I_{sim\_out} \cup I_{ref\_out} \cup \{p_{trig\_out}\}, \text{ with} \\ &\forall p_k \in I_{sim\_in} \cup I_{sim\_out}, \ p_k.ref = false \text{ and} \\ &\forall p_k \in I_{ref\_in} \cup I_{ref\_out}, \ p_k.ref = true. \end{aligned}$$

6. Optimized component groups

In order to reduce the overhead of communicating through component interfaces, we propose to group connected flexible components with the same hardware allocation into what we call a component group. The group is eventually realized as a single component with the same allocation as the grouped components.

Our solution is described through a simple example illustrated in Fig. 6. Assuming that we have two connected flexible components, C_1 and C_2, and both of them are allocated on the GPU. Without grouping, C_1 and C_2 are realized as individual components by the source-code generation; the result is graphically described in Fig. 6(a). Each component is realized to encapsulate its own device-environment through which it executes the functionality. The communication between the components functionalities is done through the components port-based interfaces.

With our approach, we optimize the realization of the two components via a component group G, as depicted by Fig. 6(b). The group contains a single device-environment through which the functionalities of both components are executed. Furthermore, the communication (i.e., data and control flow connections) between C_1 and C_2 is done at the functionality level, i.e., directly between F^{C1} and F^{C2}, by providing the output of F^{C1} as input of F^{C2}. The encapsulation of F^{C1} and F^{C2} inside G does not break the (functionality) encapsulation of C_1 and C_2.

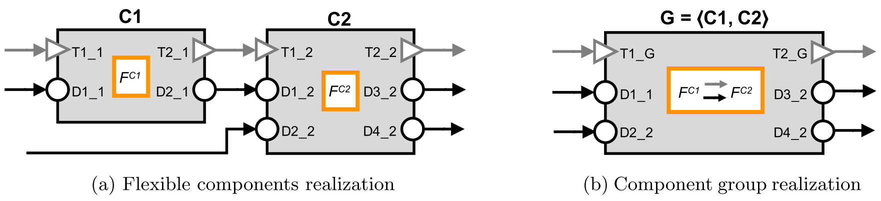
The group inherits the data ports of the enclosed components, that communicate with anything outside the group. Therefore, the group input interface contains two input data ports, i.e., *D1_1* from C_1 and *D2_2* from C_2, since these two ports receive data from outside the group. The group output interface has *D3_2* and *D4_2* data ports, both inherited from C_2. Furthermore, the group is equipped with one input and one output trigger port.

Note that a group potentially changes the overall timing of the system, by executing all components in the group together before continuing with triggered components outside the group. However, grouping is an optional optimization step, and it would be a straightforward extension to let the system developer decide which of the identified groups that should be used, and to select a suitable execution ordering among the possible ones.

The remaining of the section presents the definition of the component group and the algorithm that identifies groups in a given component-based system. For more details on the automated generation of code for the component group (i.e., code that executes the kernel functions of the flexible components in the group in the right order, manages the data transfer between kernels, and connects them to the interface of the Rubus component realizing the whole group), see Campeanu et al. (2019) and Campeanu (2018).

6.1. Definition

We define a component group G as an ordered set of connected flexible components that have the same hardware allocation (rep-



(a) Flexible components realization (b) Component group realization

Fig. 6. The transition from flexible components to component group (device-environments represented by orange boxes).

resented by *alloc(C_i)*):

$$G = \langle C_1, C_2, \ldots \rangle, \text{ where,}$$
$$alloc(G) = alloc(C_1) = alloc(C_2) = \ldots$$

Moreover, the order of the components in G defines a strict execution order, and in order for a group to be valid, we require that executing the components in this order is consistent with the triggering defined in the system.

The functionality of a component group is accessed through the port-based group interface. The interface I^G is constructed from all the ports of the grouped components that communicate with anything outside the group, as follows. An input port of the group is defined as an input port of any of enclosed components, that receives information from a component that is outside of the group. Similarly, an output port of any enclosed component that sends data to external component(s) is considered an output port of the group.

$$I^G = \{p_1, p_2, \ldots\}, \text{ where } p_m \in I^{C_1} \cup I^{C_2} \cup \ldots \cup \{p^G_{\text{trig\_in}}, p^G_{\text{trig\_out}}\}$$

The group interface is divided in two elements, one for the input ports of the group, i.e., I^G_{in}, and the other enclosing the output group ports, i.e., I^G_{out}. Furthermore, each (input and output) interface is divided in subsets according to the enclosed data types of the ports. For example, the group input interface I^G_{in} contains simple ports ($I^G_{\text{sim\_in}}$), reference ports ($I^G_{\text{ref\_in}}$) and the input trigger port ($p^G_{\text{trig\_in}}$).

$$I^G = I^G_{\text{in}} \cup I^G_{\text{out}}, \text{ where}$$

$$I^G_{\text{in}} = I^G_{\text{sim\_in}} \cup I^G_{\text{ref\_in}} \cup \{p^G_{\text{trig\_in}}\} \text{ and}$$
$$I^G_{\text{out}} = I^G_{\text{sim\_out}} \cup I^G_{\text{ref\_out}} \cup \{p^G_{\text{trig\_out}}\}$$

6.2. Group identification and connection rewiring

This section presents the algorithm that identifies the component groups in a given component-based system. We see the system as a directed graph, were each component is a node and the trigger port connection between two components is a directed vertex. In this context, identifying the groups is similar to a depth-first search algorithm starting from the clock and recursively grouping flexible components with the same allocation together. The details are presented in Algorithm 1 .

When the groups have been identified, we first go through the groups and keep only groups containing of at least two components. Next, some of the connections in the system need to be rewired. Inside a group, the connections between the enclosed components are ignored because this data communication is instead directly accomplished in the generated code for the group (for details about this code generation, see Campeanu et al. (2019) and Campeanu (2018)). Connections crossing a group boundary are rewired to the interface of the

```
function IdentifyGroups:
    $\Gamma \leftarrow \emptyset$$Visited \leftarrow \emptyset$foreach clock $C$ in the system do
        IdentifyRecursively($C$, $NULL$)
    return $\Gamma$

prodecure IdentifyRecursively($C$, $G$):
    if $C \notin Visited$ then
        add($C$, $Visited$)if $flexible(C)$ then
            if $G = NULL \vee alloc(C) \neq alloc(G)$ then
                $G \leftarrow$ createNewGroup()$alloc(G) \leftarrow$
                $alloc(C)$add($G$, $\Gamma$)
            add_last($C$, $G$)
        else
            $G \leftarrow NULL$
        foreach triggering edge$C \rightarrow C'$ do
            IdentifyRecursively($C'$, $G$)
```

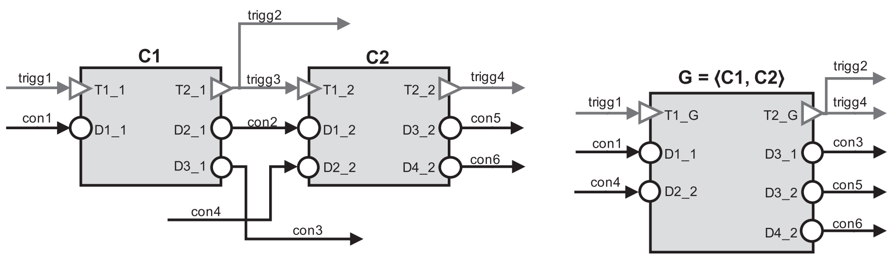
Algorithm 1: Identifying component groups.

group instead of to a component inside it. Since groups inherit the data ports of the components, data connections can remain unchanged, but all trigger connections should be rewired to the trigger ports of the group interface.

Thus, each connection from a port p_1 of component C_1 to a port p_2 of component C_2 (i.e., where $p_1 \in I^{C_1}$ and $p_2 \in I^{C_2}$), is considered by the following rules:

- If C_1 and C_2 belong to the same group, then the connection is removed.
- Otherwise, if p_1 and p_2 are data ports, the connection remains.
- Otherwise, we consider two cases (note that both cases can apply to the same pair of ports, if C_1 and C_2 belong to two different groups):
  - If C_1 belongs to a group G, then the connection is rewired so that it starts at the output trigger port of the group $p^G_{\text{trig\_out}}$ instead of starting at p_1.
  - If C_2 belongs to a group G, then the connection is rewired so that it ends at the input trigger port of the group $p^G_{\text{trig\_in}}$ instead of ending at p_2.

The connection rewiring rules are described through an example presented by Fig. 7. Two connected flexible components $C1$ and $C1$, and their connections are presented in Fig. 7a. Assuming that both $C1$ and $C1$ are allocated onto the same processing unit, their realization is optimized via a group G presented in Fig. 7b. The group inherits the data ports used for communication with components outside the group, and thus these connections remain unchanged, but connections to and from the trigger ports of $C1$ and $C1$ are rewired to the new trigger ports of the group.



(a) Two connected flexible components (b) The resulting connections

Fig. 7. The rewiring of a group ports.

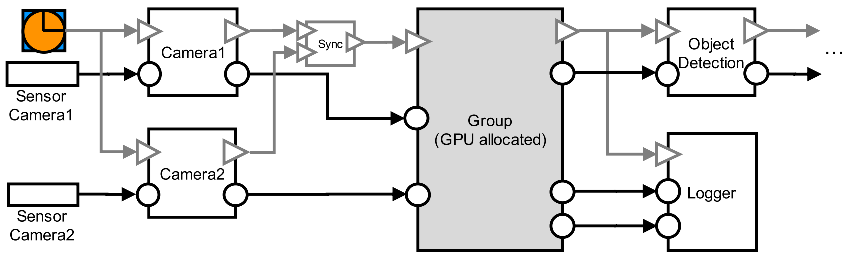


Fig. 8. Grouping and new connections in the vision system, assuming that all flexible components are allocated to the GPU.

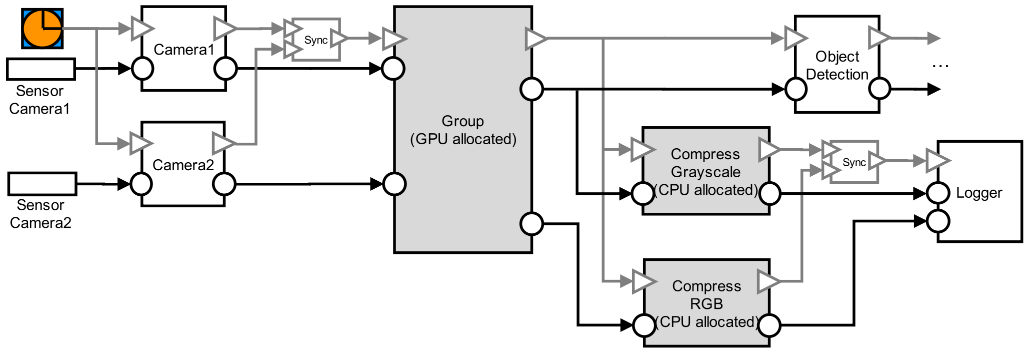


Fig. 9. Grouping and new connections in the vision system, assuming that three of the flexible components are allocated to the GPU.

In the vision system example, assuming that all flexible components were allocated on the GPU, a group is constructed as presented in Fig. 8. The group inherits two input data ports, both from *MergeAndEnhance* component, and three out output data ports from *EdgeDetection, CompressGrayscale* and *CompressRGB* respectively. The connections of the inherited data ports remain, the data connections between the enclosed components are removed, and new connections are rewired to the trigger ports of the group.

If, instead, only *MergeAndEnhance, ConvertGrayscale* and *EdgeDetection* are allocated to the GPU, the resulting grouping is shown in Fig. 9. Note that the algorithm initially also identifies two groups each containing one of the remaining components, but these singleton groups are ignored.

7. Component communication support

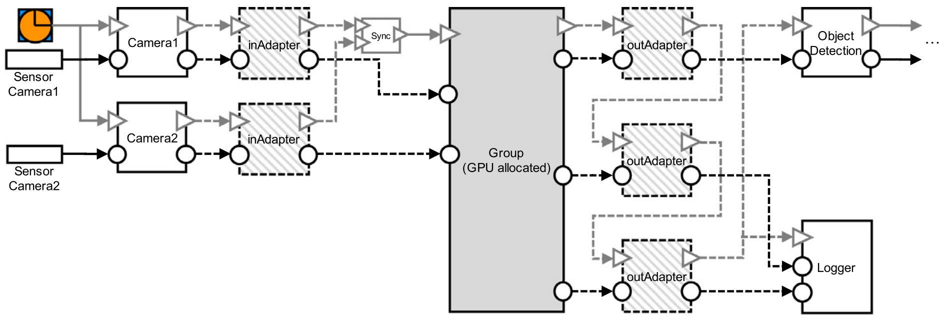
A concept that complements the component group by enabling (re-)use with no changes on different platforms, is the automatically generated *adapter*. When constructing regular components with GPU capabilities, different (data transfer) operations are required to be encapsulated inside the components. We automatically externalize these required operations in artifacts that transparently realize the communication between the groups and components of the system. In this way, the group becomes platform-independent. Below, we define how to identify where adapters are needed and how to avoid unnecessary adapters. For details on the automatic generation of adapter code (implemented as regular Rubus components), see Campeanu et al. (2019) and Campeanu (2018).

7.1. Adapter identification

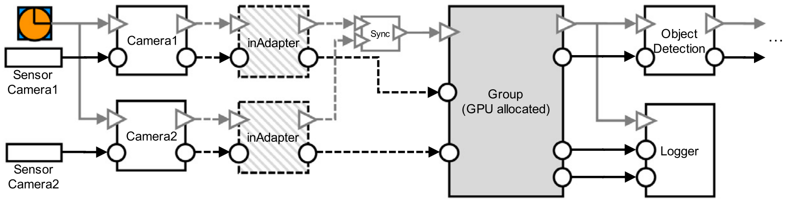
Where adapters are needed depends both on the allocation of flexible components and on the type of platform. For example, if the platform has distinct memory spaces, adapters are needed for all communication between components allocated to different units, but if the memory is shared between CPU and GPU, there is no need for adapters at all. With shared virtual memory, adapters are needed to copy data to the shared memory but there is no need to copy data back since it can be accessed also by components allocated to the CPU.

We introduce two types of adapters corresponding to the transfer activity directions, i.e., *inAdapter* and *outAdapter* adapters, each one accomplishing either a one-to-one port communication between two components, or a one-to-many port communication between several components. In our approach, adapters are implemented as regular components by following the component model specification, and are automatically generated. The following rules define the introduction of adapters and their port connections:

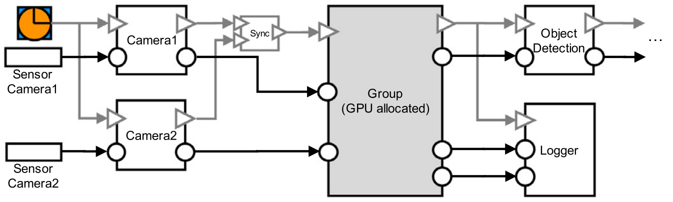
(a) Consider any two reference data ports p_1 and p_2, with a connection from p_1 to p_2, belonging to components or component groups C_1 and C_2, respectively (i.e., $\langle p_1, p_2\rangle \in Connect$, $p_1 \in I^{C_1}$, $p_2 \in I^{C_2}$ and $p_1.ref = p_2.ref = true$).
(b) Consider the following cases:
  - If the platform has a shared virtual memory, then:
    - If C_1 is a regular component and C_2 is a flexible component or a component group, than an *inAdapter* is needed.
  - If the platform has distinct CPU and GPU memories, then:
    - If C_1 is a regular component and C_2 is a flexible component or a component group, than an *inAdapter* is needed.
    - If C_1 is a flexible component or a component group and C_2 is a regular component, than an *inAdapter* is needed.
  - In all other cases, no adapter is needed.
(c) If an adapter is needed, there are two cases to consider:
  - If there is already an adapter connected to p_1, then we reuse this adapter, and just replace the original connection from p_1 to p_2 with a new connection from the output port of the existing adapter to p_2.
  - If there is no adapter already connected to p_1, then we add a new adapter with one input port connected to p_1 and one output port connected to p_2. This replaces the original connection from p_1 to p_2. Moreover, let $p^a_{trig\_in}$ and $p^a_{trig\_out}$ be the input and output trigger ports of the adapter, and $p^1_{trig\_out}$ be the output trigger port of C_1. We first add



(a) Platform with distinct memory addresses



(b) Platform with partially shared (virtual) memory



(c) Platform with fully shared memory

Fig. 10. Adapters generated for the vision system on three different platforms.

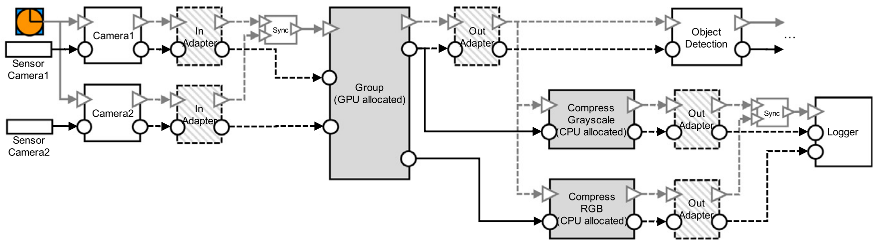


Fig. 11. Adapters generated for a platform with distinct memory addresses, assuming only three flexible components are allocated to GPU.

one connection $\langle p^1_{trig\_out}, p^a_{trig\_in}\rangle$, and then replace each connection $\langle p^1_{trig\_out}, p\rangle$ with $\langle p^a_{trig\_out}, p\rangle$.

We exemplify the adapter concept via the vision system, for the three different platform types, in Fig. 10, assuming that all flexible components are allocated to the GPU (resulting in the grouping shown in Fig. 8). The first case (a) presents the generated adapters for a platform with CPU and GPU distinct memory systems, i.e., two *inAdapter* adapters, and three *outAdapter* adapters. Next (b), the vision system is enriched only with *inAdapter* adapters, since with shared virtual memory there is no need for *outAdapter* adapters because all components can directly access the SVM space. When the platform has fully shared memory, there is no generation of adapters since all regular and/or GPU-allocate components can directly access the memory, as shown in the last case (c).

If we instead consider the case where only three flexible components (*MergeAndEnhance*, *ConvertGrayscale* and *EdgeDetection*) are allocated to the GPU (resulting in the grouping shown in Fig. 9), adapters for a platform with distinct memory systems are generated as shown in Fig. 11.

8. Evaluation

For the evaluation, we applied our approach on the vision system example. The vision system, as described by Fig. 3 (see Section 3.1), contains five components that are suitable to be executed on the GPU. To evaluate our approach, we defined five allocation scenarios for these flexible components, as shown in Table 1. In Scenario 1, all five components are allocated to the GPU, in Scenario 5 we allocate the components to the CPU, and then three additional scenarios were randomly selected where allocations alternate between CPU and GPU.

For each of the constructed scenarios, we implemented three versions of the vision system, as follows. In the naïve version (V1), each of the five components is constructed as a regular component that encapsulates all the required GPU information. In the second version (V2), the five components are seen as flexible

Table 1
Allocations scenarios for the vision system.

Flexible Component	Hardware allocation scenario				
	1	2	3	4	5
MergeAndEnhance	*GPU*	*GPU*	*GPU*	*GPU*	**CPU**
ConvertGrayscale	*GPU*	*GPU*	*GPU*	**CPU**	**CPU**
EdgeDetection	*GPU*	**CPU**	*GPU*	*GPU*	**CPU**
CompressRGB	*GPU*	*GPU*	**CPU**	**CPU**	**CPU**
CompressGrayscale	*GPU*	**CPU**	**CPU**	**CPU**	**CPU**

Table 2
The number of adapters and formed groups.

Scenario	Platform type	Number of adapters						Number of groups
		inAdapter			outAdapter			
		V1	V2	V3	V1	V2	V3	V3
1	dGPU	0	2	2	0	3	3	1
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
2	dGPU	0	2	2	0	3	3	2
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
3	dGPU	0	2	2	0	3	3	3
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
4	dGPU	0	2	2	0	3	3	5
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
5	dGPU	0	2	2	0	3	3	1
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	

dGPU – CPU and GPU distinct memory platform
iGPU1 – Shared Virtual Memory platform
iGPU2 – Shared Memory platform
V1 – Naïve version
V2 – Flexible components
V3 – Flexible components and component groups

components which are realized as regular components with their corresponding allocation. For the third version (V3), the flexible components are optimized through component groups, before being converted into regular components. Generated adapters are used for V2 and V3 versions. For each scenario, were used three different hardware platforms that contain GPUs: *(a)* a PC with an NVIDIA dGPU architecture, *(b)* an embedded platform with an AMD Kabini SoC with shared virtual memory architecture,[1] and *(c)* an embedded platform with an AMD Carrizo SoC with full shared memory architecture[1].

To examine the impact of the flexible component concept and their grouping into single entities, we compare for all three system versions:

- the size of the generated and manually written code,
- the end-to-end execution times, and
- the correctness of the output frames.

The naïve version is constructed with components that hard-code all the required GPU information. More details on the manual construction of a component with GPU capability is found in Appendix A. The manual code is written by the same researcher who implemented the code generation for adapters and component groups, meaning that they are similar with respect to coding style and that the kernel code is identical in both versions. For details on the implementation of flexible components, and the automatically generated code in adapters and component groups, see Campeanu et al. (2019) and Campeanu (2018).

Table 2 describes the created groups and generated adapters for each considered scenario and platform. In Scenario 1, when all flexible components are allocated on the GPU, depending on the used platform, there are different numbers of generated adapters. For the platform with distinct memory systems (dGPU), there are two *inAdapters* and three *outAdapters* created for the V2 and V3 versions.

In the case of the platform with virtual memory property, two *inAdapters* and no *outAdapters* are needed as the memory is directly accessed by the host and device, for both version cases (V2 and V3). For the rest of the scenarios, there are the same number of generated adapters as Scenario 1, the only difference is the allocation of the flexible components/group. In the case of the naïve version (V1), for all scenarios and on all platforms, there is no generated adapter given that the transfer activities are encapsulated inside the components.

Regarding the created groups, in Scenario 2 there are two created groups where one, that is allocated on the GPU, encloses the connected components *MergeAndEnhance, ConvertGrayscale* and *CompressRGB*, while the other, allocated on the CPU, contains *EdgeDetection* and *CompressGrayscale*. In Scenario 3 there are three created groups, where one group, allocated on the GPU, encloses *MergeAndEnhance, ConvertGrayscale* and *EdgeDetection*, while the other two components enclose single components, i.e., *CompressRGB* and *CompressGrayscale*. Given that there are no connected components with the same allocation, Scenario 4 contains five created groups, each enclosing single components. The last scenario is similar to the first one with one created group, enclosing all components, that is allocated on the CPU.

Table 3 presents the generated and manually written code, and the total used code for all five scenarios, platforms and system versions. Regarding the manually written characters, the naïve version contains the highest number, with a maximum of 14,909 characters for the dGPU type of platform. While the characteristics of the platform change, the manual written code is changing given that e.g., there are different requirements for data transfer activities. For platforms with virtual memory (iGPU1), where there is no need for transferring data from device to host, there are written 13,474 characters. For the last type of platform with a share memory system (iGPU2), due to the fact that there is no need of data transfer activities, a number of 11,961 characters are required.

Regarding the generated characters, the number of generated adapters and groups (see Table 2) influence the generated code size of the vision system. We notice that for shared memory platforms (dGPU), the flexible component-based version (V2) has 9007 generated characters that includes two *inAdapters* and three *outAdapters*. For the same system version, in the case of the platform with virtual memory (iGPU1), the number of generated characters drops to 6852 because there are no *outAdapters*. Finally, for shared memory platform (iGPU2) where there are no required adapters, there are 5414 generated characters. In the case of the component group-based version (V3), there are less characters generated than V2 version due to the enclosing of components into group entities. For all scenarios and platforms, there are no generated characters for the naïve version as all the code is manually written.

For the second part of the evaluation, Table 4 presents the execution time for all three versions of the system, in all five scenarios. We mention that the execution time was computed for the entire system, from the starting of the execution of *Camera1* and *Camera2* until *ObjectDetection* and *Logger* finished their execution. Furthermore, the experiments were done on a machine with 2,6 GHz i7 processor and two distinct memory systems (dGPU type of platform).

[1] https://unibap.com/product/advanced-hetereogeneous-computing-modules/

Table 3
The code size of the system versions.

Scenario	Platform type	Code size (# characters)								
		Generated			Written			Total		
		V1	V2	V3	V1	V2	V3	V1	V2	V3
1	dGPU	0	9007	8816	14909	1720	1720	14909	10727	10536
	iGPU1	0	6852	6759	13474	1720	1720	13474	8572	8479
	iGPU2	0	5414	5223	11961	1720	1720	11961	7134	6943
2	dGPU	0	9007	8953	14909	1720	1720	14909	10727	10673
	iGPU1	0	6852	6796	13474	1720	1720	13474	8572	8516
	iGPU2	0	5414	5360	11961	1720	1720	11961	7134	7080
3	dGPU	0	9007	8945	14904	1720	1720	14909	10727	10665
	iGPU1	0	6852	6788	13474	1720	1720	13474	8572	8508
	iGPU2	0	5414	5352	11961	1720	1720	11961	7134	7072
4	dGPU	0	9007	9007	14904	1720	1720	14909	10727	10727
	iGPU1	0	6852	6852	13474	1720	1720	13474	8572	8572
	iGPU2	0	5414	5414	11961	1720	1720	11961	7134	7134
5	dGPU	0	9007	8816	14909	1720	1720	14909	10727	10536
	iGPU1	0	6852	6759	13474	1720	1720	13474	8572	8479
	iGPU2	0	5414	5223	11961	1720	1720	11961	7134	6943

dGPU – CPU and GPU distinct memory platform
iGPU1 – Shared Virtual Memory platform
iGPU2 – Shared Memory platform
V1 – Naïve version
V2 – Flexible components
V3 – Flexible components and component groups

Table 4
Execution times of the system versions on the dGPU platform.

	End-to-end execution time (msec)		
Scenario	V1	V2	V3
1	15.06	13.86	13.10
2	30.75	29.91	29.60
3	27.83	21.45	20.84
4	33.25	25.52	25.10
5	45.36	36.17	33.87

V1 – Naïve version
V2 – Flexible components
V3 – Flexible components and component groups

For the naïve version (V1), we notice that for Scenario 1, where all components are executed on the GPU, the execution time is the fastest (15 msec), while for Scenario 5 where all components are allocated on CPU, the execution time is the slowest (45 msec). In the case of the virtual memory system (V2), its execution time is faster than the naïve version on all platforms due to the usage of the adapters. In Scenario 1, the execution time of V2 is faster with 1.2 ms than V1, while in Scenario 5, V2 is faster with 9.2 msec than V1. In the case of the component group-based version (V3), we notice that the execution is faster than V2 in all scenarios, but the difference is small except for scenario 5.

Regarding the results, we compared the three produced frames (i.e., one input frame of *ObjectDetection* and the two input frames of *Logger*) by the three versions, in all five scenarios and for all types of platforms. The conclusion was that the generated frames were identical in all cases and the introduced solutions (e.g., adapters) did not modify the correctness of the output.

9. Related work

The latest technological progress has facilitated the development of Systems-on-Chip (SoC) with multiple heterogeneous processors (e.g., CPU, GPU, FPGA) into a single chip. In this sense, Andrews et al. propose the usage of COTS components to address SoC systems with CPUs and FPGAs (Andrews et al., 2004). The authors developed, based on the multithreading POSIX programming paradigm, an interface abstraction layer to ease the component synchronization over the shared memory. The work targets embedded applications composed of components that concurrently execute and synchronize and exchange data. Although the work focuses on creating a model for CPU-FPGA systems, the authors point out that the existing way to develop applications, i.e., developing components specific to the CPU/FPGA, is opposite to the desired goals (e.g., modularity, portability, reuse). In the context of component development of systems with GPU capability, we introduce the work of Bernier et al. (2017). The authors present a way of developing, using the OpenCL framework, military radio applications that use platforms with GPUs. The work describes how components encapsulate the data transfer activities, which makes them platform specific and greatly affecting the reusability aspect.

Regarding systems with GPU capabilities, we mention a general-purpose component model called PEPPHER (Dastgeer et al., 2012) that proposes a way to efficiently utilize CPU-GPU hardware. In this sense, the authors define the PEPPHER component as an annotated software unit. The component interface is described by an XML document that contains the name, parameter types and access type of the component. Dealing with platforms with different processing units (i.e., CPUs and GPUs), the interface may define several implementation variants of the same component. Regarding the data passed between PEPPHER components, this is wrapped into portable and generic data structures called *smart containers*. These structures ensure the data transfer between the memories of the processing units. In our work, we provide a transparent and automatic way of transferring data between processing units by using adapters. Similar to the smart container concept which is more complex when dealing with memory management, our adapters can be considered as high-level memory management elements. Moreover, regarding the different PEPPHER component variants, our work's advantage is that it has less overhead (e.g., memory footprint, development time) by using a single flexible component. After the component is allocated to either the CPU or GPU, automatic means generate the appropriate variant of the flexible component.

We also mention the Global Composition Framework (GCF) that extends the PEPPHER component model (Dastgeer and Kessler, 2013). Although GCF focuses on performance optimization, the work addresses the development of systems with GPUs. The considered component model defines the notion of component that consists of an interface. The interface describes a functionality and the functionality is implemented by multiple variants. The interfaces and implementation variants have attached meta-data (e.g., to target GPU). While in PEPPHER, the meta-data is described by an XML, under GCF, pragmas are used to represent the meta-information. The data used by a component needs to reside in the memory space associated with the component's execution platform. The framework takes care of the data transfer between components that reside in disjoint memory spaces, using runtime support. This is done via a static data analyzer that checks the program data flow in order to find out where to place data handling code. The runtime support allows the data transfer only when an implementation gets called, minimizing the memory overhead. However, it is difficult to use runtime assistance in the domain targeted in our work given that a transfer that is unknown when is realized, may present a big overhead for a control-type of system.

Another work that targets heterogeneous systems with e.g., CPU, GPU and FPGA is the Elastic computing framework (Wernsing and Stitt, 2012). Although is not a component model per se, this framework has similar principles (e.g., interfaces, usage of already developed functions). The framework uses a library that contains the so called *elastic functions*. An elastic function has different implementations. For example, an implementation may use the CPU and GPU, while another uses only the CPU. The framework, being focused on the system performance, analyzes the execution time of the elastic functions for all combinations of resources, and decides during run-time, the fastest implementation for a given combination of resources. The Elastic framework handles resource allocation and data management inside the elastic function. An improvement provided by our work is that it externalizes the data management outside the component (i.e., using adapters), in this way enhancing the component reusability.

Brock et al. extend an existing library (i.e., PVTOL) in order to support GPU architecture (Brock et al., 2012). The authors take in consideration multiple memory architectures which are handled through *conduits*. Basically, a conduit is a template, where, based on its arguments, it targets the desired platform (i.e., CPU or GPU). The extended library has similarities with our work. While, in Brock's work, the developer introduces, at the source-code level, the conduits to address the desired platform, in our work we leave this choice to the designer, that, after the application architecture is constructed, decides which platform to be use for the flexible components. Another aspect targeted by Brock et al. is the memory management which is also done via conduits, implemented by the developer. In our work, (part of) the memory management is done automatically via the adapter artifacts.

Another work that uses a library to address the GPU capabilities in mathematics, is introduced by Winter et al. (2014). The library converts data-parallel expressions into kernel functions written using the PTX programming language (i.e., an assembler language which is intermediary between high-level CUDA C/C++ code and GPU machine code). Furthermore, the library automates the memory access via a *cache* and *page-out* mechanisms. The cache mechanism extracts required data fields and make them available to the GPU, which are paged-out (i.e., copied to the CPU) when they are accessed by the CPU or there is not enough memory to hold other cached data fields. Due to the narrow focus (i.e., mathematics) of this library, the mechanism that automatically generates kernel functions cannot apply to our work. Furthermore, the used PTX language increases the demands on the developer to posses knowledge in (low-level) GPU development. Regarding the memory management, the *page-out* mechanisms cannot be utilized in our work given the introduced unpredictability of data transfer activities.

Other works use a different approach, i.e., model-driven engineering, for development of SoC embedded systems. For example, Gamatie et al. (2011) present the GASPARD design framework for massively parallel embedded systems. Designing the systems at a high abstraction level using the MARTE standard profile, the GASPARD framework allows the designers to automatically generate code of high-performance embedded systems. Another work that is worth mentioning is the of work of Rodrigues et al. (2013). The authors extend the MARTE profile to allow modeling of GPU architectures. This work, as well as the GASPARD framework, introduce mechanisms to handle the GPU memory system and its interaction with the main memory system. Being developed in 2011, both frameworks focus on systems only with distinct (CPU and GPU) memory systems. In our work, we cover newer platforms with GPUs that have different characteristics regarding the memory access. The components proposed in our work, complemented by the adapter mechanisms, allow the transparent usage of many of the existing embedded platforms with GPUs.

10. Summary and conclusions

Existing CBD frameworks offer no specific support for the construction of embedded systems with GPUs. In this context, the existing approaches introduce several shortcomings which diminish the benefits of using CBD in embedded systems with GPUs. The main goal of this paper is to facilitate the component-based development of embedded systems with GPUs, by introducing specific GPU mechanisms as follows. A *flexible component* is provided, which can be executed with no modifications required, on either CPU or a variety of GPU platforms. The required information to allow the component to be executed on the selected platform is automatically generated. An optimization step is also introduced, were connected flexible components that share the same executable platform (i.e., CPU or GPU) are grouped in entities that behave as regular components. To facilitate the communication between components, we introduce artifacts called *adapters* to handle data transfer activities between connected components. The adapters are automatically generated when needed, depending on the platform characteristics and on which executable platform components are placed.

Through the introduction of these specific mechanisms, we facilitate the development of embedded systems with GPUs and provide the following benefits:

- Introducing a type of component that can be executed on different processing units, without manual modification, increases the *flexibility* of designing embedded systems.
- The flexible component is platform-independent, being capable to be executed on platforms with GPUs that have different characteristics. For example, the same component can be executed, with no modification required, on platforms that have distinct CPU and GPU memory systems but also on platforms that fully share the same memory. Moreover, rather than hard-coding a GPU configuration inside the component, this can be provided later during the system development. This increases the component *reusability*.
- Due to the mechanisms that automatically generate the required information which the component requires to be suc-

cessfully executed on the selected platform, the development of components with GPU capability is simplified. The adapter is another factor that decreases the development effort due to the fact that the developer is not responsible anymore to handle specific data transfer activities.
- Component *maintainability* is also increased by the adapters. Through this concept, the specific data transfer activities are not encapsulated inside the component anymore, but automatically handled.

The proposed concepts are implemented by extending a state-of-practice component model, namely the Rubus component model, as follows. The flexible components are realized as regular Rubus components, characterized by interfaces, constructors, behavior functions and destructors. Similarly, we implemented component groups into single regular Rubus components and defined corresponding optimization rules. Adapters were also realized as regular Rubus components. Rubus was also used to construct a vision system, which was used to evaluate the feasibility of the proposed approach to facilitate component-based development of embedded systems with GPUs.

Several future work directions remain to continue our work. As there are various types of heterogeneous embedded systems such as Spartan 6 equipped with FPGA,[2] ways to apply and/or extend our work on these types of systems can be considered. FPGA is a processing unit that excels in parallel processing data, being an alternative to the GPU. Due to the fact that an FPGA is equipped with a private memory system, the adapters artifacts may be adapted to facilitate the communication between components allocated on CPU and FPGA. Furthermore, given that the OpenCL framework provides also support for this type of systems, the implementation of the adapters artifacts may be directly use for this type of embedded systems.

In this work, we consider platforms with uni-core CPUs and GPUs. Given that many of the modern SoC systems that integrate GPUs contain multi-core CPUs, another future direction may address mechanisms that allow the multi-cores to simultaneous use the GPU. In this sense, the GPU is seen as a shared resource, and mechanisms should also protect it to not be over-used. For example, the multi-cores can simultaneous use the GPU as long as they require less or equal processing threads as the GPU has available.

As the GPU is a parallelizible processing unit, a future work direction includes the parallel scheduling of components with GPU capability. For example, if the hardware has much resources, several components with GPU capability may be run in parallel. This future work direction may complemented the previous one, where multi-cores simultaneous access the GPU and execute, in parallel, their functionality. In the same direction, and related to the grouping of flexible components with the same allocation into single entities, we want to investigate mechanisms for parallel execution of component groups. Such mechanisms should analyze which functionalities are independent of each other and permit their parallel execution, according to the platform resources.

Declaration of Competing Interests

None.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research (SSF) through the RALF3 project (IIS11-0060); and the Knowledge Foundation in Sweden through MINEStrA (20170133) and ORION (20140218). We also want to thank the anonymous reviewers for valuable insights and suggestions how to improve the presentation of the work.

Appendix A. A component with GPU capability

Below, we present the construction of the *ConvertGrayscale* component, one of the components in the running vision system example that benefit from using the GPU. Listing 1 illustrates the code that is manually written to construct the behavior function of the *ConvertGrayscale* component when implemented as a regular Rubus component. In comparison, when implemented as a flexible component, the component behavior is defined by just the kernel function, corresponding to lines 2–25 in the manual version, and the rest of the code is automatically generated as part of adapters and component groups.

The component receives a 2D color frame via the *ID_input1* port, processes it and provides an output frame in gray scale format. The actual functionality of the component, referred as the kernel function, is described from line 1 to 25, where the function receives the input frame (i.e., via the pointer parameter *in*) and its properties (i.e., the *width* and *height* parameters), and provides an output frame (i.e., via the *out* parameter). Inside the function, the individual colors of a pixel are accessed by the current processing thread (determined using the *index* position), through three different variables that are initialized with their corresponding values (lines 18, 19 and 20). Finally, each output pixel is initialized with its gray scale value (line 25).

Setting up the platform and defining the required mechanisms are presented between lines 28 and 45. For example, at line 39, a queue mechanism is constructed; this will be used to send instructions to the device. The program code continues with the creation of two memory buffers:

- The *input* buffer is created (line 48) on the device (e.g., GPU) to contain the data received as input. The data is copied from the host into the created buffer using the *clEnqueueWriteBuffer* function, as presented at line 54.
- The *result* buffer is created (line 51) on the device (e.g., GPU) to hold the data resulted from the kernel execution.

The kernel object construction (line 57) and the values of its parameters (line 60–63) are defined. The parameters are initialized with the values received via the input port. For example, the $args \rightarrow IP.ID\_input1 \rightarrow height$ construction allows to access the height property of the inputed frame. The settings regarding the GPU threads are defined at lines 66 and 67, followed by the execution of the kernel at line 70. Once the result is computed, a memory buffer is created on the host (line 73) and the result of the kernel execution is copied (line 76). In the last part of the kernel, instructions to clean up are defined, such as the release of the created queue mechanism (line 85).

[2] https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html

Listing 1
The code encapsulated by *ConvertGrayscale*

```
const char *source =
"__kernel void grayscale(__global const unsigned char *in, const int width, const int
     height, __global unsigned char *out) \n"
"{ \n"

" /* compute absolute image position (x, y) */ \n"
" int row = get_global_id(0); \n"
" int col = get_global_id(1); \n"

" /* relieve any thread that is outside of the image */ \n"
" if(row >= width || col >= height) \n"
" return; \n"

" /* compute 1-dimensional pixel index */ \n"
" int index = row + width * col; \n"

" /* load RGB values of pixel (converted to float) */ \n"
" float3 pixel; \n"
" pixel.x = in[3 * index]; \n"
" pixel.y = in[3 * index + 1]; \n"
" pixel.z = in[3 * index + 2]; \n"

" /* compute luminance and store to output array */ \n"
" float lum = 0.2126f*pixel.x + 0.7153f*pixel.y + 0.0721f*pixel.z; \n"

" out[index] = (unsigned char)lum; \n";

/* Get platform and device information */
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint num_devices;
cl_uint num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_devices);

/* Create an OpenCL context */
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, NULL);

/* Create a command queue */
cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, NULL);

/* Create a program from the kernel source */
cl_program program = clCreateProgramWithSource(context, 1, (const char **)&source, NULL
     , NULL);

/* Build the program */
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

/* Create memory buffer on the device to hold the input frame */
void *input = apiCreateBuffer(context, CL_MEM_READ_WRITE, 3*(args->IP.ID_input1->width)
     *(args->IP.ID_input1->height) * sizeof(unsigned char), NULL, NULL);

/* Create memory buffer on the device to hold the output result */
void *result = apiCreateBuffer(context, CL_MEM_WRITE_ONLY, (args->IP.ID_input1->width)*(
     args->IP.ID_input1->height) * sizeof(unsigned char), NULL, NULL);

/* Copy the input image to its respective memory buffer */
clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, 3*(args->IP.ID_input1->width)*(
     args->IP.ID_input1->height) * sizeof(unsigned char), args->IP.ID_input1->ptr, 0,
     NULL, NULL);

/* Create the OpenCL kernel */
cl_kernel krn = clCreateKernel(program, "grayscale", NULL);

/* Set the arguments of the kernel */
```

(*continued on next page*)

Listing 1 (*continued*)

```
SetKernelArg(krn,0,sizeof(cl_mem),(void*)&input);
SetKernelArg(krn,1,sizeof(int),(void *)&(args->IP.ID_input1->width));
SetKernelArg(krn,2,sizeof(int),(void *)&(args->IP.ID_input1->height));
SetKernelArg(krn,3,sizeof(cl_mem),(void *)&result);

/* Setup the GPU settings */
size_t global[2] = {600, 450};
size_t local[2] = {8, 8};

/* Execute the OpenCL kernel */
clEnqueueNDRangeKernel(command_queue, krn, 2, NULL, global, local, 0, NULL, NULL);

/* Create memory buffer on the host to hold the resulted frame */
unsigned char *h_result = (unsigned char*)malloc(sizeof(unsigned char)*(args->IP.
     ID_input1->width)*(args->IP.ID_input1->height));

/* Transfer the memory buffer result on the device */
clEnqueueReadBuffer(command_queue, result, CL_TRUE, 0, (args->IP.ID_input1->width)*(args
     ->IP.ID_input1->height) * sizeof(unsigned char), h_result, 0, NULL, NULL);

/* Clean up */
clFlush(command_queue);
clFinish(command_queue);
clReleaseKernel(krn);
clReleaseProgram(program);
clReleaseMemObject(input);
clReleaseMemObject(result);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```

References

Andrews, D., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., Ortiz, J., Komp, E., Ashenden, P., 2004. Programming models for hybrid FPGA-CPU computational components: a missing link. IEEE Micro 24 (4), 42–53.

AUTOSAR – Technical Overview, 2019. http://www.autosar.org. Accessed: 2019-04-24.

Bernier, S., Lévesque, F., Phisel, M., Zvernik, D., Hagood, D., 2017. Using OpenCL to increase SCA application portability. J. Signal Process. Syst. 89 (1), 107–117.

Box, D., 1997. Essential COM, 1st Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Brock, J., Leeser, M., Niedre, M., 2012. Heterogeneous tasks and conduits framework for rapid application portability and deployment. In: Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012). IEEE, pp. 1–9.

Bures, T., Carlson, J., Crnkovic, I., Sentilles, S., Vulgarakis, A., 2008. ProCom – The Progress Component Model Reference Manual, version 1.0. Technical Report. Mälardalen University, Sweden.

Campeanu, G., 2018. GPU Support for Component-based Development of Embedded Systems. Mälardalen University, Sweden.

Campeanu, G., Carlson, J., Sentilles, S., 2017. Developing CPU–GPU embedded systems using platform-agnostic components. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications.

Campeanu, G., Carlson, J., Sentilles, S., 2017. Flexible components for development of embedded systems with GPUs. In: 24th Asia-Pacific Software Engineering Conference.

Campeanu, G., Carlson, J., Sentilles, S., 2018. Optimized realization of software components with flexible OpenCL functionality. In: 13th International Conference on Evaluation of Novel Approaches to Software Engineering.

Campeanu, G., Carlson, J., Sentilles, S., 2019. The realization of flexible GPU components in Rubus. Technical Report. Mälardalen University.

Campeanu, G., Carlson, J., Sentilles, S., Mubeen, S., 2016. Extending the Rubus component model with GPU-aware components. In: Component-Based Software Engineering (CBSE), 2016 19th International ACM SIGSOFT Symposium on. IEEE, pp. 59–68.

Crnkovic, I., Larsson, M., 2002. Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA.

Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M., 2011. A classification framework for software component models. IEEE Trans. Softw. Eng. 37 (5), 593–615.

Dastgeer, U., Kessler, C., 2013. A framework for performance-aware composition of applications for GPU-based systems. In: Parallel Processing (ICPP), 2013 42nd International Conference on. IEEE, pp. 698–707.

Dastgeer, U., Li, L., Kessler, C., 2012. The PEPPHER composition tool: performance-aware dynamic composition of applications for GPU-based systems. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. IEEE, pp. 711–720.

Gamatie, A., Le Beux, S., Piel, E., Ben Atitallah, R., Etien, A., Marquet, P., Dekeyser, J.-L., 2011. A model-driven design framework for massively parallel embedded systems. ACM Trans. Embedded Comput. Syst.(TECS) 10 (4), 39.

Hänninen, K., Mäki-Turja, J., Nolin, M., Lindberg, M., Lundbäck, J., Lundbäck, K.-L., 2008. The Rubus component model for resource constrained real-time systems. In: Industrial Embedded Systems, 2008. SIES 2008. International Symposium on. IEEE, pp. 177–183.

Helmerich, A., Koch, N., Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., et al., 2005. Study of worldwide trends and R&D programmes in embedded systems in view of maximising the impact of a technology platform in the area. Final Report for the. European Comission, Belgium.

John, K.H., Tiegelkamp, M., 2010. IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids. Springer Science & Business Media.

Ke, X., Sierszecki, K., Angelov, C., 2007. COMDES-II: a component-based framework for generative development of distributed real-time control systems. In: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007). IEEE, pp. 199–208.

Liggesmeyer, P., Trapp, M., 2009. Trends in embedded software engineering. IEEE Softw. 26 (3), 19–25.

Lowy, J., 2005. Programming. NET Components: Design and Build. NET Applications Using Component-Oriented Programming. O'Reilly Media, Inc..

Manavski, S.A., 2007. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: International Conference on Signal Processing and Communications. IEEE, pp. 65–68.

OMG CORBA Component Model, 2019. http://www.omg.org/spec/CCM/4.0/. Accessed: 2019-04-24.

Preis, T., Virnau, P., Paul, W., Schneider, J.J., 2009. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. J. Comput. Phys. 228 (12), 4468–4477.

Rodrigues, A.W.O., Guyomarc'h, F., Dekeyser, J.L., 2013. An MDE approach for automatic code generation from UML/MARTE to OpenCL. Comput. Sci. Eng. 15 (1), 46–55.

Sun Microsystems, 2019. JavaBeans Specifications. https://www.oracle.com/technetwork/java/docs-135218.html.Accessed: 2019-04-24.

The OpenCL specification version 2.1, 2019. https://www.khronos.org/developers/reference-cards/. Accessed: 2019-04-24.

Wernsing, J.R., Stitt, G., 2012. Elastic computing: a portable optimization framework for hybrid computers. Parallel Comput. 38 (8), 438–464.

Winter, F., Clark, M.A., Edwards, R.G., Joó, B., 2014. A framework for lattice QCD calculations on GPUs. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, pp. 1073–1082.

Gabriel Campeanu received his PhD from Mälardalen University, Sweden, in 2018, with a thesis titled GPU Support for Component-based Development of Embedded Systems. He currently works at Bombardier Transportation.

Jan Carlson is a professor in computer science, specializing in software engineering at Mälardalen University, Sweden. His current research focuses on component- and model-based development of embedded systems, addressing areas such as architectural decision support, allocation optimization, model-level timing analysis and code generation. He also works on real-time analysis, in particular focusing on shared stack usage and response time analysis under limited preemption scheduling.

Séverine Sentilles is a Senior Lecturer in the Industrial Software Engineering research group at Mälardalen University, Sweden. Her research interests include component-based development and model-based development and their use to improve the support of extra-functional properties in the development of embedded systems.