Facilitating Component Reusability in Embedded Systems with GPUs

Gabriel Campeanu

Mälardalen Real-Time Research Center,
Mälardalen University, Västerås, Sweden
Email: gabriel.campeanu@mdh.se

Abstract. One way to fulfill the increased requirements (e.g., performance) of modern embedded systems is through the usage of GPUs. The existing embedded platforms that contain GPUs bring several challenges when developing applications using the component-based development methodology. With no specific GPU support, the component developer needs to encapsulate inside the component, all the information related to the GPU, including the settings regarding the GPU resources (e.g., number of used GPU threads). This way of developing components with GPU capability makes them specific to particular contexts, which negatively impacts the reusability aspect. For example, a component that is constructed to filter 640x480 pixel frames may produce erroneous results when reused in a context that deals with higher resolution frames. We propose a solution that facilitates the reusability of components with GPU capabilities. The solution automatically constructs several (functional-) equivalent component instances that are all-together used to process the same data. The solution is implemented as a state-of-the-practice component model (i.e., Rubus) and the evaluation of the realized extension is done through the vision system of an existing underwater robot.

1 Introduction

Modern embedded systems deal with huge amount of information that usually originates from the interaction with the environment. For example, the Google autonomous car[1] receives from its various sensors (e.g., camera, LIDAR, radar, ultrasound) an amount of 750 MB of data per second. This data is processed with enough performance in order for the car to be coordinated with the environment changes, such as moving pedestrians.

The embedded boards with Graphics Processing Units (GPUs) are feasible solutions for tackling the stringent requirements of modern embedded systems. Through its thousands of computational threads, the GPU is more efficient than the CPU when dealing with parallel data processing. For instance, a stereo matching application developed for embedded systems, delivers an increased frame rate processing when is executed on the GPU [9].

[1] https://waymo.com

Another trend in the embedded systems domain is the usage of the component-based development (CBD) for construction of systems [5]. This software engineering methodology promotes the development of systems through the composition of already existing software units called software components. CBD is successfully adopted by industry through various component models such such as AUTOSAR [13], Rubus [8] and IEC 611-31 [10].

The existing component models for embedded systems development offer no specific GPU support. One way to develop components with GPU capabilities, is to encapsulate inside the components, all the GPU-related information. This leads to constructing components that are specific to particular contexts. A challenge appears when (re-)using the same component (that has particular GPU specifications encapsulated inside) in different applications. For instance, assuming we have a component that converts color frames into the grayscale format and has encapsulated a number of 640x480 GPU threads to use (i.e., one thread per pixel). When this component is (re-)used in applications that deal with 1024x960 pixel color images, it may result in providing erroneous results.

In this work, we provide a solution to increase the reusability of components with GPU capabilities, by constructing multiple instances of the same component in order to handle data of any size specification. For example, our solution generates three more instances of a component that filters 640x480 pixel frames in order to handle images with 2560x1920 pixels. The solution divides, via specific artifacts, an initial input data into several parts that are independently handled by the generated component instances. Based on the application design, the required artifacts and component instances are automatically generated into code during the system generation stage. We implement our solution as an extension of an existing industrial component model (i.e., Rubus) and evaluate it using an existing underwater robot case study.

The remainder of the paper is divided as follows. The background information covering GPUs and embedded system is presented in Section 2. The problem description (Section 3) and solution overview (Section 4) are presented using a running-case example. Section 5 describes the solution realization, while its evaluation is enclosed in Section 6. After we describe the related work (Section 7), we conclude the paper with a discussion section.

2 GPUs in embedded systems

The two main environments to develop GPU applications are CUDA[2] and OpenCL[3]. While CUDA is developed by NVIDIA to specifically address their hardware (i.e., NVIDIA GPUs), OpenCL is a general framework that addresses different processing units (e.g., mCPUs, GPUs, FPGAs) produced by various vendors. In this work, due to the fact that most of the existing embedded platforms with GPUs (developed by e.g., Intel, AMD, NVIDIA, Altera, IBM,

[2] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[3] https://www.khronos.org/opencl/

Samsung, Xilinx) support OpenCL, we utilize this particular environment for development of GPU functionality.

When using OpenCL to create GPU functionality, the developer needs to construct several hierarchical steps, as follows. At the highest level is the platform that contains the drivers. The platform addresses the existing devices, such as the GPU and the CPU. One of the device should be selected to execute the functionality. Memory buffers should be allocated in order to hold the input and/or output data of the functionality. The functionality, also know as the *kernel* may be constructed at any time, regardless of the required hierarchical steps. The arguments of the kernel are defined using the allocated input and output buffers. The number of threads used to execute the functionality are specified in such a way to match the platform characteristics, i.e., to not exceed the available thread resources. Furthermore, the threads are organized in particular groups (e.g., tiles of 8x8 threads) that directly influence the performance of the kernel execution.

Nowadays, the GPU is successfully integrated on various embedded platforms. Various vendors such as Intel, NVIDIA, Xilinx and AMD provide different embedded solutions with different characteristics, suitable for different domains. For example, while the NVIDIA Condor GR2[4] is utilized in high-performance solutions due to its high resources, the Mali-470 GPU with a low computation and energy consumption is employed in the construction of smart watches.

Component-based development is a software engineering methodology that promotes the development of applications through the composition of existing software units called components [5]. A core principle of CBD is the (re-)use of components in different contexts, which enhances the development efficiency. The communication between components is done through *interfaces*, which are specifications of the components' access points. In our work, we use *port-base* interfaces for sending/receiving data of different types.

An important aspect of CBD is the *encapsulation* fundamental, where all the component data and operations are encapsulated inside and hidden from anything outside. The only way to access the encapsulated information is through the component access points referred as *interfaces*. The way a component is constructed (alongside with its interfaces) is specified by a *component model*. Moreover, the component model defines the manner in which components communicate with each other, when composed into a system [4].

CBD is successfully adopted in industry through various component models. For real-time and embedded systems, the domain which we focus in this work, component models such as AUTOSAR [13], Rubus [8] and IEC 611-31 [10] are currently used in the development of applications. Particular interaction styles are employed by these component models when used for particular type of applications [6]. For example, the *pipe-and-filter* style, which is considered in this work, is suitable for streaming-of-events type of applications and adopted by e.g., Rubus and IEC 611-31 component models [6].

[4] http://www.eizorugged.com/products/vpx/condor-gr2-3u-vpx-rugged-graphics-nvidia-cuda-gpgpu/

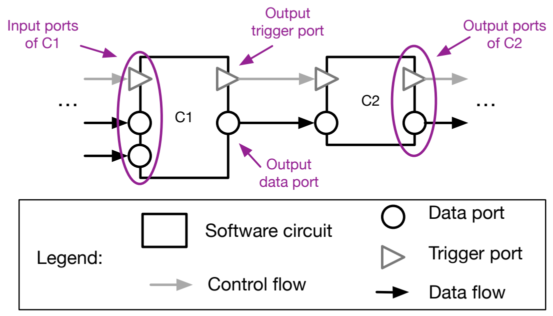


Fig. 1: Two connected Rubus components

As the Rubus component model is used in this work to realize our solution and for evaluation purposes, we introduce more details regarding it. The name of a Rubus component is *software circuit*. Each component is equipped with one input and output trigger port and one or several input and output data ports. Through these distinct types of ports (i.e., control and data ports), the component model provides a separation between control and data flow of the system, which allows an easy mapping between the control specifications of real-time and embedded systems, and the interaction model. Fig. 1 illustrates two connected Rubus components characterized by input and output (trigger and data) ports. A Rubus component follows the Read-Execute-Write execution model. For example, *C1* is initially in an idle state. After receives the control of execution through its input trigger port, it Reads the input information via its two input data ports, Executes its functionality and Writes the result in the out data port. Finally, it passes the execution control to *C2* through the output trigger port and re-enters in the idle mode.

3 Problem description

The existing component models used in the development of embedded systems provide no specific GPU support. Accordingly, the component developer, when constructing components with GPU functionality, needs to encapsulate all the GPU-related information inside the components. In this work, we focus on the encapsulated GPU settings regarding the GPU computation resources. For instance, for a component that filters frames of 640x480 pixels, the developer needs to hard-code inside the component a number of 640*480 GPU threads, where each thread processes a pixel. Moreover, the specified threads need to be grouped in a particular way (e.g., tiles of 8 by 8 threads) in order to match the size of the processed data. The grouping settings have a direct impact over the system performance.

A challenge comes when a component that contains hard-coded GPU settings is (re-)used in different contexts. Due to its encapsulated settings, the component may produce erroneous results when dealing with data of different characteristics. For example, assuming there is a component that filters images and has encapsulated settings corresponding to 640x480 pixel frames. When the component is (re-)used in an application that handles 1024x1024 pixel images, it would be able to process only a part of the system data. An alternative would be to construct a component encapsulating the same functionality, but different GPU settings corresponding to 1024x1024 pixel images. Constructing components that can be used only in certain contexts, would significantly decrease the benefits of adopting CBD for construction of embedded systems with GPUs.

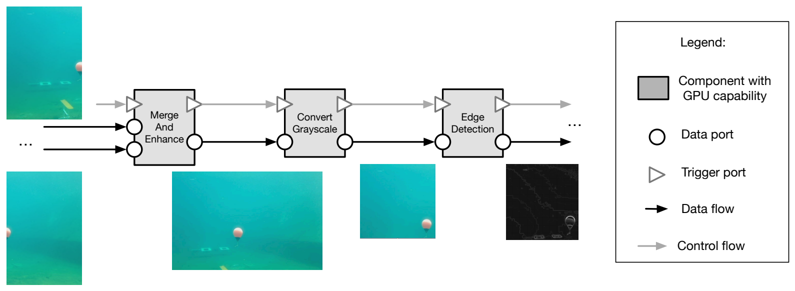


Fig. 2: Fragment of the component-based vision system of an underwater robot

To exemplify the challenge discussed in this work, we introduce a running-case example. We use a part of the vision system of an underwater robot, as described by Fig. 2. The vision system is constructed using the Rubus component model. The underwater robot is equipped with two cameras which provide a continuous flow of frames. Each pair of frames is received by the *MergeAndEnhance* component that merges and reduces the frames' noise. The resulted merged frame is converted to a grayscale format by *ConvertGrayscale* component and forwarded to *EdgeDetection* component that outputs a black-and-white frame, where objects are delimited by while lines.

We assume that the components were reused from different applications, and they were initially constructed with different GPU settings, as follows:

- *MergeAndEnhance* processes two 300x400 pixel frames and produces one 600x400 pixel frame,
- *ConvertGrayscale* converts 300x300 pixel frames, and
- *EdgeDetection* filters 600x500 pixel frames.

In the current example, the physical cameras provide frames with 300x450 pixels. The flow of the frames through the system including the (erroneous) outputs are illustrated in Fig. 2. We notice that the output of *EdgeDetection* is actually a portion of the (merged) frame produced by the *MergeAndEnhance* component.

4 Solution overview

In order to facilitate reusability of components with GPU functionality in different contexts, we provide the following solution. Based on the design information, a component is instantiated with a number of identical instances in order to process data of various sizes. Each instance independently handles a part of the original data. The output processed data from all the instances are automatically merged together into a single one that the component communicates to its connected components.

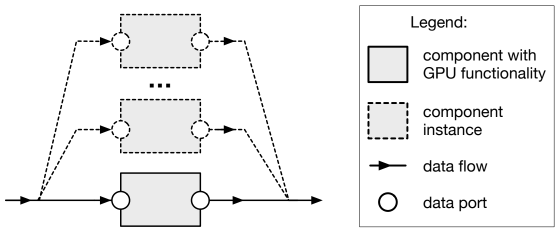


Fig. 3: Overview of the proposed solution

The overview of the proposed solution is illustrated in Fig. 3, where a component is duplicated into several more instances (illustrated in dashed lines) in order to handle income data with characteristics that are not supported by the component. The proposed solution automatically:

– decides on how many instances are needed to correctly process the corresponding data,
– distributes data to each instance, and
– gathers the outcome of each instance into a single output data.

The instances are automatically generated, in a transparent way.

Using the vision system running case, we describe our solution applied on the *ConvertGrayscale* component, as illustrated in Fig. 4. The solution automatically constructs three more component instances in order to correctly process the input received frame. The output grayscale frames provided by the used instances are merged together into a single frame which represents the conversion output of the *ConvertGrayscale* component.

5 Realization

In this work, we use white-box components, i.e., components with readable source code such as Rubus and IEC 61131 components. Furthermore, we target component models that follow the *pipe-and-filter* architecture style due to the embedded systems targeted by our solution (e.g., real-time, control-type applications). In this context, we provide a solution to automatically facilitate the

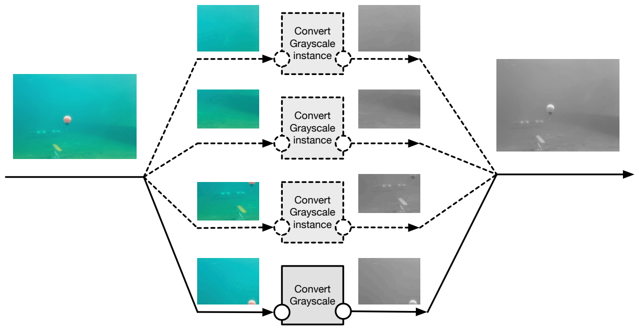


Fig. 4: Instances of the ConvertGrayscale component

(re-)use of components with GPU capability in different contexts. The solution is implemented as an extension of a state-of-the-practice component model (i.e., Rubus). Moreover, we use the OpenCL environment to implement components with GPU capability.

For each component with GPU capability, our solution checks the application design, i.e., the characteristics of the input ports and their received data. When a mismatch appears, the solution computes the number of required instances to handle the received data, and realize them inside the component, using the existing Rubus generation rules.

In order to divide the data, we introduce an artifact called *fork*. When a component is instantiated (i.e., differences between the income data attributes and component capabilities) a *fork* element is created. In order to not introduce additional component model elements, we use the existing Rubus framework and realize the *fork* artifact as a regular component equipped with input and output (trigger and data) ports. Based on the number of the input data ports of a component with GPU capability, the connected artifact will be equipped with an appropriate number of (input and output) data ports to handle all the data connections.

Similarly, the system generates an artifact called *join* to gather the outcomes from all component instances into one single outcome. Based on the number of output data ports of the component with GPU capabilities, the *join* artifact will be equipped with an appropriate number of ports to carry out the component communication. The *join* component is realized as a Rubus component, in an automatic and transparent manner. The *fork* and *join* are generated for each component with GPU capabilities, when needed.

Our solution handles the re-wiring between the introduced *join/fork* artifacts and the component and its created instances. The existing Rubus rules regarding component wiring are modified, and we introduce new rules that link the interfaces of the *join/fork* components with the generated interfaces of the component instances, and rest of the system. Moreover, in order to not introduce additional overhead for the system designer, the introduced components and their connections are transparently generated.

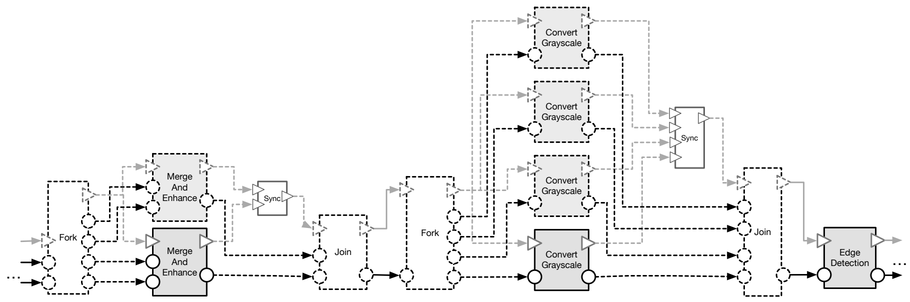


Fig. 5: The solution realization of the vision system

Fig. 5 describes the solution realization of the vision system running-case. For the *MergeAndEnhance* component, a *fork* component is automatically generated and divides the input data into two parts which are provided to the *MergeAndEnhance* component and its instance. In order to connect to the two component instances[5], the *fork* component is equipped with:

– two input data ports that correspond to the number of *MergeAndEnhance* input data ports, and
– four output data ports through which it provide data to the two connected instances.

To gather all the outcomes, a *join* component is generated to copy the two results into one single location. The *join* component has:

– two input data ports, where each one receives the output data of the two generated component instances, and
– one output data port through which it sends the (gathered into one) result to the *ConvertGrayscale* component.

In a similar way, a *fork* component is created to divide and provide the correct data to the four *ConvertGrayscale* instances, and a *join* component to gather all the four results into a single frame. The system does not need to create any *fork/join* components for *EdgeDetection* due to its specifications, i.e., its functionality can handle the inputed data frame.

5.1 Implementation

The solution, based on the design of the system, constructs the proposed artifacts at the generation system stage, presented in the following paragraphs.

A Rubus software component is characterized by a header and several C source files that contain the definitions and declarations for:

[5] for simplification, we use the term of *instances* to refer to a component and its instance(s)

– an interface that contains structures used to define the input and output data ports,
– a constructor that initiate the resource requirements of the component,
– a behavior function that defines the component functionality, and
– a destructor that releases the allocated resources of the component.

The same Rubus rules that generate regular components (with GPU capability) are followed when implementing the component instances. For example, the implementation of the *MergeAndEnhance* component is identical with its instance.

In the next paragraphs, we describe the implementation of the *join* and *fork* components. The existing Rubus rules regarding generation of component interface are used to implement the interface of *fork* elements, as follows. Located in the header file, the interface contains a structure declaration that is composed of two (structure) elements corresponding to the input and output data ports. Fig. 6 presents the interface of the fork component, i.e., *SWC_fork_MergeAndEnhance* (lines 34-37). The interface contains two elements:

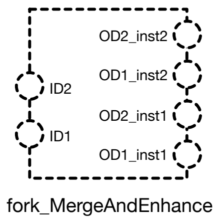
– *IP_SWC* - a structure with two elements that correspond to the input data ports, and
– *OP_SWC* - a structure with four elements corresponding to the output data ports of the fork component.

The two input ports receive the input data of *MergeAndEnhace* component, i.e., data locations and width and height dimensions of the input images. The output ports will contain, after the execution of the behavior function, the details of the data sent to each of the component instance. In a similar way, the interface of the *join_MergeAndEnhance* component will be automatically constructed.

In general, we let the system to generate the constructor for the *fork* component as it does not have distinctive requirements. On the other hand, the *join* component constructor needs special attention due to the artifact purpose, i.e., to copy several data into one location. Therefore, the *join* constructor needs to manage the allocation of a memory space to hold all of the combined data. This is realized with the OpenCL function *clCreateBuffer*[6] that allocates memory space on the GPU. The generation of the *join_MergeAndEnhance* constructor is described in Listing 1.1, where the width and height sizes of the outcome image are calculated using the corresponding values of the frames received through each input port (i.e., lines 3 and 8). The computed width and height sizes are used to allocate a memory space (i.e., line 11) that is big enough to contain all of the received data.

The behavior function of a *fork* component, based on the information of the input data and the number of connected component instances and their characteristics, computes the characteristics (i.e., location, width and height) of the data that will be processed by each component instance. For the *fork_MergeAndEnhance*

[6] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html



```
typedef struct {
  GPU_unsigned_char *ptr;
}data_type;

typedef struct {
  int size;
}dim1;

typedef struct {
  int size;
}dim2;

typedef struct {
  data_type *place;
  dim1 width;
  dim2 height;
}img_format;

/* input ports */
typedef struct {
  img_format *ID1;
  img_format *ID2;
}IP_SWC;

/* output ports */
typedef struct {
  img_format OD1_inst1;
  img_format OD2_inst1;
  img_format OD1_inst2;
  img_format OD2_inst2;
}OP_SWC;

/* the interface declaration */
typedef struct {
  IP_SWC IP;
  OP_SWC *OP;
}SWC_fork_MergeAndEnhance;
```

Fig. 6: The interface of a *fork* component

component, the behavior function calculates two pointers that direct to two memory locations corresponding to two parts of the initial image. The output data ports are set with the computed pointers, alongside with the corresponding image dimensions (i.e., width and height).

Listing 1.1: The constructor of a *join* component

```
int width = 0;
<foreach input port p>
  int width += args->IP.p->width.size;
<endforeach>

int height = 0;
<foreach input port p>
  int height += args->IP.p->height.size;
<endforeach>

void *location = clCreateBuffer(contex, CL_MEM_WRITE_ONLY, 3*width*height, NULL, NULL);
```

The behavior function of the *join* component copies all the income data into a single memory location (allocated by the constructor). For our vision system example, the *join_MergeAndEnhance* behavior function copies two images using the

OpenCL function *clEnqueueCopyBuffer*[7]. The specifications of the input data (determined through the input port characteristics) and the memory location (allocated by the constructor) to hold the two images, are used as parameters for the copying activity. Listing 1.2 describes one of the copy activities done by the behavior function, i.e., the copy of the image received through the input port ID1. Similarly, another copy activity corresponding to the data received by the second input port, is generated inside the behavior function.

Listing 1.2: A part of the behavior function of a *join* component

```
clEnqueueReadBuffer(command_queue, args->IP.ID1->place->ptr, (unsigned char*) location, 0, 0, 3*(args->IP.ID
      ->width.size)*(args->IP.ID->height.size) * sizeof(unsigned char), 0, NULL, NULL);
```

Regarding the destructors, we do not include specific instructions for the *fork* destructor and let the system to handle it using the existing Rubus framework rules. For the *join* component, the destructor needs to release the memory allocated by the constructor. Therefore, for the generation of *join_MergeAndEnhance* destructor, we use the *clReleaseMemObject*[8] function to release the memory allocated by the constructor, as depicted in Listing 1.3.

Listing 1.3: The destructor of a *join* component

```
clReleaseMemObject(location);
```

6 Evaluation

In this section, we examine the feasibility of our solution using the introduced running case, i.e., the vision system of the underwater robot. Moreover, we analyze: *i)* the execution overhead, and *ii)* memory overhead introduced by the generation of the *fork* and *join* components, and the component instances.

We constructed two Rubus versions of the vision system, where one is constructed using our solution and the other, referred as the custom version, is constructed using regular components with hard-coded settings to explicitly handle the system input images. In all of the experiments, we make comparisons between the two versions. Furthermore, the GPU functionality of each component in both versions is constructed in such a way to handle also input frames that have lower (width and height) attributes than the component specifications. The platform used for the experiments in an embedded platform with an AMD Kabini SoC [9].

Listing 1.4 presents a part of the *ConvertGrayscale* functionality. For simplification purposes, we define three variables (i.e., lines 1-3) to hold the attributes of the input data (i.e., location, width and height). These variables are used as arguments for the *ConvertGrayscale_fct* function (referred as the GPU kernel)

[7] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueCopyBuffer.html
[8] https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/clReleaseMemObject.html
[9] https://unibap.com/product/advanced-hetereogeneous-computing-modules

that contains the conversion-to-grayscale GPU functionality. The hard-coded settings, that correspond to the GPU thread index, are accessed from the kernel function using specific calls (i.e., *get_global_id*) in lines 10 and 11, and checked against the received kernel arguments (i.e., line 12). When the arguments values do not match (i.e., less than) the number of utilized threads, the extra threads are discarded (i.e., they do not execute).

Listing 1.4: Part of the *ConvertGrayscale* GPU functionality

```
1   unsigned char* in = args->IP.ID1->place->ptr;
2   int width = args->IP.ID->width.size;
3   int height = args->IP.ID->height.size;
4
5   __kernel void ConvertGrayscale_fct(__global const unsigned char *in, int width, int
         height, __global unsigned char *out)
6   {
7   /* compute absolute image position (x, y) */
8    int x =  get_global_id(0);
9    int y =  get_global_id(1);
10
11  /* relieve threads that are outside of the received image */
12   if (x >= width || y >= height) return;
```

The execution of the two vision system versions produced results (i.e., frames) that were identical. The introduced artifacts and activities (i.e., dividing and merging data) did not influence the system's outcomes.

In the second experiment, we focused on the execution time overhead of the introduced solution. Our solution introduces: *i)* four artifacts (i.e., *fork* and *join* components), *ii)* one additional *MergeAndEnhance* instance, and *iii)* three more *ConvertGrayscale* instances. To examine the introduced overhead, we calculate the end-to-end execution of the two vision system versions. The execution time for the version implemented with our solution was 9.4 msec, while the custom version had 4.6 msec.

As the memory characteristic is a sensible topic in the embedded systems domain, in the last part of the evaluation we analyzed the memory overhead introduced by our solution. For the custom version of the vision system, the total memory requirement is of 494.2 kB, where *MergeAndEnhance* requires 177.6 kB of memory, *ConvertGrayscale* requires 175.6 kB of memory, and *EdgeDetection* requires 130.8 kB of memory. For the version that uses our solution, where there are two *MergeAndEnhance* instances, four *ConvertGrayscale* instances and one *EdgeDetection* instance, the total system requirement is of 722 kB of memory. We mention that a *MergeAndEnhance* instance that processes two 300x400 pixel frames requires 158.6 Kb of memory and a *ConvertGrayscale* instance that processes (at a time) one 300x300 pixel frame, requires 63.7 kB of memory. We notice that the custom version requires with 227 kB less memory than the version constructed with our solution.

Although the custom version has an improved execution time and requires less memory than the version that uses our solution, the components are specifically constructed for this application and platform, and have a low reusability in other (software and hardware) contexts.

7 Related work

The increased requirements of modern applications lead to the adoption of heterogeneity in embedded systems. AUTOSAR component model, utilized in the automotive industry, was extended with multi-core ECUs support [11]. Another solution to satisfy the increased demands of modern applications is to use accelerator hardware. The FPGA is one of the feasible solutions to be used as co-processor for data demanding applications. Andrews et al. proposes a way to use COTS components, referred as hybrid components, to develop applications for CPU-FPGA hardware [1].

The GPU in the context of embedded systems is addressed by Campeanu et al. which facilitate the development of applications for CPU-GPU hardware [3]. The authors proposed to enrich each component (with GPU capability) with a specific configuration interface. Through this interface, the component receives from e.g., the system designer, individual GPU settings regarding the number of GPU threads used by the functionality. We consider this as a possible solution to tackle the challenge discussed in this work, but it comes with two disadvantages, as follows. The system developer needs to have, at the time when designing the system, detailed information (i.e., the physical GPU threads limitation) of the hardware platform that will host the applications. The detailed hardware platform is not always known at design time. Another downside is that the system designer needs to: *i)* have knowledge about GPU development, or *ii)* correspond with the component developer (breaking the separation-of-concern CBD principle), in order to provide good/best GPU thread settings for the components with GPU capability. Although there is an overhead introduced by our solution (i.e., memory usage and execution time), we believe that our work increases the reusability aspect while preserving the separation-of-concern principle.

Some component models develop traditional (CPU-based) systems by hard-coding inside the components the specific characteristics of the hardware. Lednicki et al. introduce an additional layer (i.e., mapping layer) to address the flexibility of components [12]. The introduced mapping layer connects software and hardware platforms allowing them to be independently developed. We consider that new architectural elements (e.g., software layers) would greatly increase the overhead of Rubus solutions. Therefore, we constructed our solution using the existing elements of the Rubus framework.

It is worth to mention the work of Dastgeer et al. that introduce the PEPPHER framework that uses a component-based development approach to construct CPU-GPU systems [7]. They refer to a software component as a block that encapsulates one or several implementation variants. All the variants are computationally equivalent and have the same interface. Whenever the component is executed, a proper variant is selected based on the software call parameters and available hardware resources. In our work, we also use multiple instances of the same component; however, all of the instances are used in order to produce the correct output.

8 Discussion

The existing component models have no specific GPU support in development of embedded systems. This leads to constructing components, with hard-coded settings, that are specific to certain contexts. To tackle this challenge, we propose a solution to improve the reusability of components with GPU capability, for different contexts. The solution introduces two types of artifacts, i.e. *fork* and *join* artifacts, that are automatically generated to divide and merge data. Moreover, the solution instantiates each component with GPU functionality with a number of instances in order to handle data of any size.

There are other solutions (e.g., see Section 7, second paragraph) to tackle the challenge discussed in this work. As presented in the evaluation, a component with GPU capability may be hard-coded with (high number of) GPU threads settings to handle images of different sizes. There are two limitations of this solution, as follows:

- each GPU platform has a limited number of threads (e.g., CL_DEVICE_MAX_WORK_GROUP_SIZE for OpenCL contexts). Hard-coding large thread resources inside components will make them specific to certain GPU platforms, with large available resources.
- Imposing large GPU thread usage for components with GPU capability used to process data of small-to-medium sizes, leads to waste of GPU resources.

Another downside of the proposed solution is the memory overhead. For each component that cannot handle the received data, we generate a number of instances, alongside the *join* and *fork* components. The vision system version implemented with our solution has generated four more component instances (i.e., one for *MergeAndEnhance* and three for *ConvertGrayscale*) and four *fork/join* components. We consider that the memory overhead is a major downside of our solution, due to the domain targeted of our work (i.e., real-time and embedded systems) where the system is characterized by limited resources (e.g., available memory). Therefore, one future work direction is to reduce the introduced memory overhead by grouping all the created instances in a conceptual component. Another future work direction is to increase the GPU parallelism level by simultaneous executing the instances of the same component. Using the existing approaches for parallel execution of components on GPU [2], we may improve the system performance, while delivering an increased component reusability.

Besides the overhead introduced by our solution (i.e., increased memory usage and execution time), we consider that our solution decreases the existing gap of component-based development for embedded systems with GPUs, facilitating the reusability of components with GPU capabilities.

Acknowledgment

The work is partially supported by the Knowledge Foundation through the ORION project (reference number 20140218).

References

1. D. Andrews, D. Niehaus, and P. Ashenden. Programming models for hybrid CPU/FPGA chips. *Computer*, 37(1):118–120, 2004.
2. G. Campeanu. Parallel execution optimization of GPU-aware components in embedded systems. In *The 29th International Conference on Software Engineering & Knowledge Engineering SEKE 2017, 5-7 Jul 2017, Pittsburgh, United States*, 2017.
3. G. Campeanu, J. Carlson, S. Sentilles, and S. Mubeen. Extending the Rubus component model with GPU-aware components. In *Component-Based Software Engineering (CBSE), 2016 19th International ACM SIGSOFT Symposium on*. IEEE, 2016.
4. M. Chaudron and I. Crnkovic. Component-based software engineering. In H. van Vliet, editor, *Software Engineering: Principles and Practice*. Wiley, 2008.
5. I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.
6. I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *IEEE Transaction of Software Engineering*, 37, October 2011.
7. U. Dastgeer, L. Li, and C. Kessler. 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, 2012.
8. K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback. The Rubus component model for resource constrained real-time systems. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*. IEEE, 2008.
9. M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze. A fast stereo matching algorithm suitable for embedded real-time systems. *Computer Vision and Image Understanding*, 2010.
10. K.-H. John and M. Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.
11. F. Kluge, M. Gerdes, and T. Ungerer. AUTOSAR OS on a message-passing multicore processor. In *SIES*, pages 287–290, 2012.
12. L. Lednicki, J. Feljan, J. Carlson, and M. Zagar. Adding support for hardware devices to component models for embedded systems. In *The Sixth International Conference on Software Engineering Advances*, 2011.
13. A. D. Partnership. Technical overview v4.2. `http://www.autosar.org`, (accessed April 14, 2018).