Extending the Rubus Component Model with GPU-aware Components

Gabriel Campeanu, Jan Carlson, Séverine Sentilles and Saad Mubeen
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
Email: {gabriel.campeanu, jan.carlson, severine.sentilles, saad.mubeen}@mdh.se

***Abstract*—To support the complex functionality expected of modern embedded systems, the trend is to supplement CPUs with Graphical Processing Units (GPUs), thus creating heterogeneous embedded systems. However, taking full advantage of GPUs increases the complexity of the development and requires dedicated support, and no such support exists in the component models currently available. The only solution today is to completely encapsulate all GPU-specific information and operations within the components, which increases the communication overhead and reduces component reusability, thus significantly limiting the applicability of component-based development to heterogeneous embedded systems.**

In this paper, we propose to extend Rubus, an industrial component model for embedded systems, with dedicated support for GPUs. We introduce new constructs, including GPU ports and automatically generated adapters to facilitate seamless component communication over the heterogeneous processing units, regardless of whether the components use the CPU, GPU, or both. A running example is used for the problem description, the proposed extension, and to evaluate our solution.

I. Introduction

During the last decades, embedded systems have rapidly spread in almost all types of areas and domains. Nowadays, embedded system applications provide a wide range of services in industry, healthcare, automotive and military domains. The trend of modern applications such as autonomous vision-based robots [1] or vehicle vision systems [2] is to include more complex, resource-demanding functionalities, also in embedded systems with limited resources in terms of computational power. A solution to these challenges is achieved through the use of modern parallel systems with, for instance, a combination of CPU-GPU computation units. Compared to the sequential execution model of CPUs', GPUs bring the benefit of faster processing of large amount of data thanks to their ability to parallelize data processing. Being constructed with a parallel processing architecture that has a private memory system, specific information and operations are required to use the GPU hardware. The use of GPUs also increases the complexity of software development even further, because dedicated programming models must be used, such as CUDA [3] and OpenCL [4].

One way to alleviate the complexity of software development is through component-based development (CBD), where applications are constructed by composing already existing software components [5] [6]. The CBD approach has been successfully used in academia with component models such as SOFA [7] or Palladio [8]. Acknowledged component models such as AUTOSAR [9] or Rubus [10] are also used in industry for developing embedded system applications. However, when developing applications for CPU-GPU embedded systems, there is no explicit support in the existing component models to help with GPU development. Currently, in order to develop GPU-based applications, all the GPU specific information and operations have to be encapsulated within the software components, which *1)* causes unnecessary communication overhead between components; and *2)* reduces the component reusability as each component requiring the GPU must use dedicated computation settings (e.g., number of GPU computation threads). For example, a component internally setting the number of GPU threads can only be reused on hardware devices that have enough computation resources to support this number, and would not be able to take advantage of any additional resources.

In our previous work [11], we outlined an approach to address these drawbacks by introducing high-level explicit GPU modelling support in CBD. The approach relies on introducing new concepts that mitigate the currently inadequate component-based development for GPUs: the concepts of GPU ports and adapters are suggested to facilitate the component communication mechanism. We also proposed a way to improve the component reusability by enabling to configure, at system-level, the GPU computation settings for each component using the GPU.

In this paper, we further develop the envisioned approach and explore how to concretly integrate it into a component model that is successfully used in industry, namely the Rubus component model. The idea is to introduce the required elements with low impact on the component model's specification, in order to reduce the impact on existing timing analysis techniques and tool chains. This is achieved by extending Rubus with the concept of GPU-aware component which is derived from standard Rubus component enhanced with dedicated GPU elements (e.g. GPU ports and configuration interface).

The rest of the paper is organized as follows. Background information about Rubus and GPU-based embedded systems are provided in Section II. The problem statement is presented, using a running example, in Section III, while the

approach overview is described, continuing the same example, in Section IV. The implementation of the Rubus extension is presented in Section V. Section VI covers our evaluation of the introduced extension. Related work is described in Section VII, and the paper is concluded in Section VIII.

II. Background: Rubus and GPUs

In this section, essential concepts of the Rubus component model and GPUs are introduced.

A. Rubus

The Rubus component model [10] focuses on development of embedded systems with real-time properties. It aims to cover data-flow software architectures for systems with one or several processing units. The lowest hierarchical level component in Rubus is called software circuit (SWC), and is characterized by input and output ports, which define the interface of the component. The behavior of a component is specified in one or several entry functions, implemented in C.

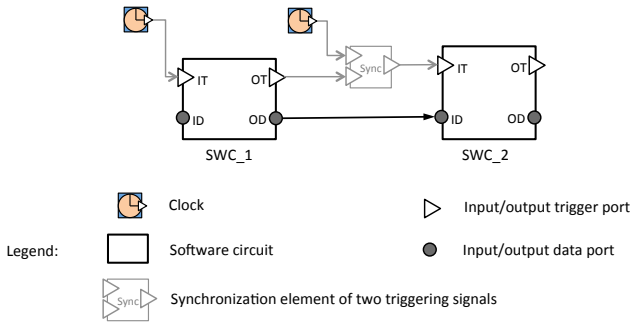


Fig. 1: Rubus software circuits

As mentioned before, a software circuit has input and output ports. There are two categories of ports: data ports used to carry out the system data flow, and trigger ports that realize the control flow in the system. A Rubus component has only one input and one output trigger port, but can have many input and output data ports. Fig. 1 presents two components, each with an input trigger port (IT), and output trigger port (OT). The components receive the data through the input data port ID, and provide the result through the output data port OD. All the data ports are characterized by a type. A data output port of one SWC can be connected to one or several data input ports that have the same data type. Similarly, a trigger output port of one SWC can be connected to one or several trigger input ports of other SWCs. A trigger input port can be activated by either a component, a clock (element that periodically provides triggering signals), a sporadic event or an interrupt. A Synch element is used to synchronize two trigger signals as is exemplified in the figure.

The execution semantics of a component are Read, Execute, Write. Initially, the component is in an inactive mode until its input trigger port is triggered. When triggered, the component switches to the Read mode where it starts reading data from its input data ports. Based on the read data, the component functionality is performed in the Execute mode. After the execution is completed, the result is written in the output data ports during the Write execution mode. Finally, the output trigger is activated, after which the component returns to the inactive state.

B. GPUs

A distinct characteristic of a GPU is that it cannot be independently used without a CPU. The CPU, considered as the brain of the system, triggers all the GPU activities. Being a processing unit with a private memory system, the data needs to be shifted onto the GPU random access memory (RAM) system and then used in the processing activities. When the GPU activities finish, the result produced by the GPU needs to be shifted back onto the main RAM to be used in the rest of the system functionality. The shifting operations are done by specific procedures provided by GPU API programming models, such as the *cudaMemcpy* function implemented by the CUDA API [3].

Whenever the CPU initiates an activity onto the GPU, it also needs to specify how much of the GPU computation resources (i.e., threads) the activity uses during its execution. The available resources of the GPU hardware should be able to execute the specified configuration settings; if not, the activity cannot be executed.

Unlike CPUs, GPUs are constructed with hundreds of execution cores that can handle multiple parallel and independent calculations. Hence, a system can harness the execution power of a GPU and use it for demanding and heavy data parallel processing activities. The bottleneck of having a GPU with a distinct memory system is the overhead of copying data between the CPU and GPU.

III. Problem description

In this section we describe the current approach using existing component models to develop applications which require GPU computation. Each component needs to encapsulate all the specific GPU information and operations in order to address the hardware platform. Whenever a component uses the GPU hardware to execute its functionality, it needs to shift its required input data from the main memory system (i.e., the CPU RAM system) onto the GPU memory system, (i.e., GPU RAM system). After finishing its processing activities, the result also needs to be shifted back onto the main memory system. Therefore, specific transfer operations and memory initialization mechanisms need to be enclosed in the component. In addition, when accessing the GPU hardware, the component needs to specify its computation settings, including how much of the hardware computation thread resource should be used for the processing activity. These GPU computation settings are thus also embedded into the component code.

In its current state of specifications, the Rubus component model needs to follow the same approach when addressing GPU-based hardware. Each SWC which requires GPU computation, needs to encapsulate all the GPU-specific information

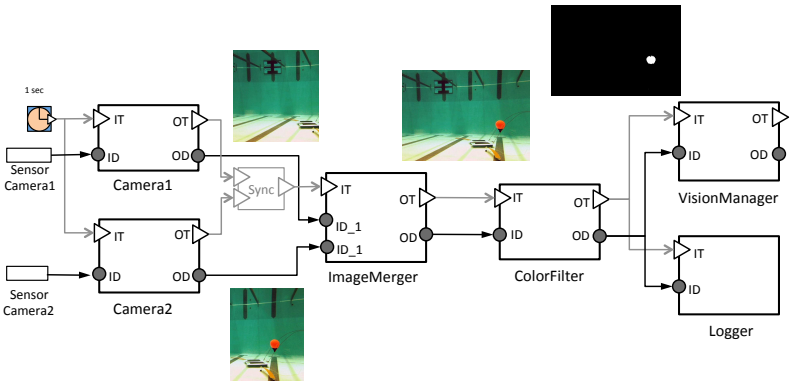


Fig. 2: Component-based architecture of the Vision System demonstrator in Rubus

and operations, such as data transfer operations or the GPU computation settings.

In order to discuss the problem in more details, we consider the following example that presents a part of a component-based design of a demonstrator modeled with the Rubus component model. The demonstrator, developed at Mälardalen University, Sweden, is an underwater robot that autonomously navigates under water, executing various missions such as tracking red buoys [12]. The robot is equipped with two cameras which provide a continuous stream of image frames to an embedded electronic board composed of a CPU and a GPU, where both units have different memory systems.

Fig. 2 illustrates the Rubus design of the robot's vision system. Two camera SWCs are connected to the physical cameras. Each component processes one image each, resulting two images at the same time. The frames are forwarded to the *ImageMerger* SWC which checks the overlapping frame area and merges the frames into one single frame utilizing the GPU hardware. The merged frame is delivered to the *ColorFilter* SWC which filters the colors of the image on the GPU, and produces a black-and-white image result. The filtered frame is analyzed by the *VisionManager* SWC which takes appropriate actions based on the shape, size and position of the detected object. The filtered frame is also sent to a *Logger* component, that registers the robot's navigation data.

Both *Camera1* and *Camera2* SWCs access the main RAM system to acquire the raw image frames, process them on the CPU and finally store them back onto the RAM. Having GPU computation, the hardware activities of the *ImageMerger* SWC are different. In addition to the RAM access to obtain the cameras' frames, the component needs to copy them onto the GPU memory system to merge them. Once the merging procedure finishes, the component copies the result back onto the RAM system. Fig. 3 describes the detailed activities of the vision system.

In the general case, when using the Rubus component model to develop applications that requires GPU usage, the following challenges are faced:

- An inefficient inter-SWCs communication. Using the existing Rubus communication mechanism, the SWCs with GPU computation are compelled to copy the (input and output) data from and to the RAM system. This produces a communication overhead over the CPU-GPU hardware connection, which degrades the performance of the system.
- Duplicated software code. As an effect of encapsulating the same data transfer operations, there is a lot of duplicated code in the system.
- A reduced reusability of the components. When accessing the GPU, each SWC needs to specify how much of the GPU computation resources (i.e., threads) are utilized. This information is hard-coded within the component. Hence, the reuse of the SWC is limited to specific hardware platforms that posses enough GPU computation resources to allow the execution of the component.

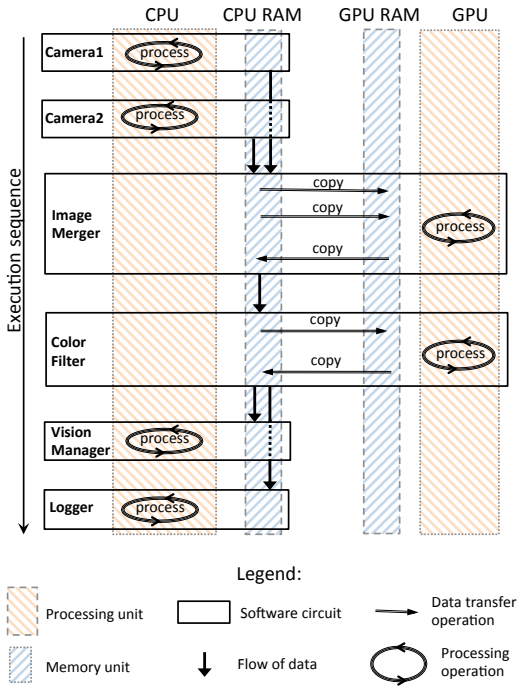


Fig. 3: The hardware activities of the vision system

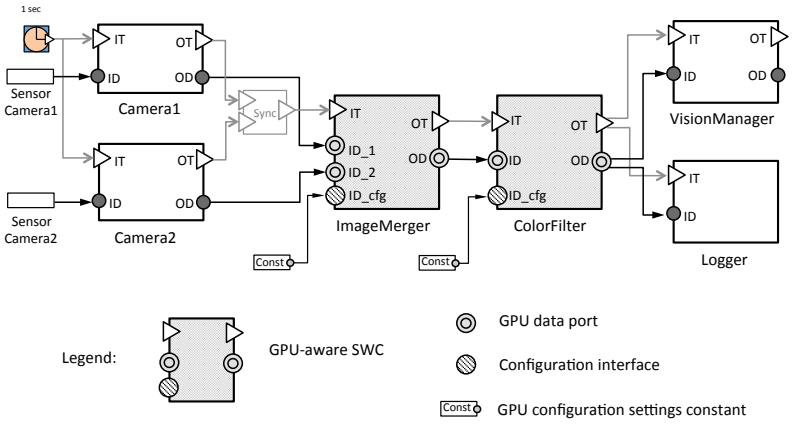


Fig. 4: The demonstrator vision system design using the extension

IV. The Rubus Extension

One way to tackle the current shortcomings in the Rubus component model is to introduce new component model elements, as follows:

- GPU ports. These ports are aware of the GPU environment. They facilitate a direct communication of GPU-based components, through the GPU RAM system.
- Connection adapters. Whenever a regular data port is connected to a GPU port or vice-versa, adapters are automatically generated to facilitate the copy operation between the different processing units.
- A configuration interface. Through it, the system developer distributes suitable GPU computation settings to GPU-based component. These settings correspond to the computation resources of the hardware platform in use.

To integrate our solution into the Rubus component model, we add the following changes to it. We introduce a new type of component, i.e, the GPU-aware software circuit, that is characterized by GPU computation. A GPU-aware SWC is characterized by:

- A new category of Rubus data port called GPU data port. Being aware only of the GPU memory system, these ports have a different type than the regular data ports. The difference is justified by the fact that the GPU hardware may have a different memory architecture than the architecture of the main memory system. For example, an embedded system may have a 64-bit architecture for the main memory and a 32-bit architecture for the GPU memory. When the data transfer between the two different memory systems is done, the data should have appropriate types, with respect to the corresponding memory architecture. GPU ports can be connected to both regular data ports and GPU data ports. They are mandatory elements when constructing a GPU-aware component.
- A configuration interface to improve the component reusability. Through it, the system distributes to SWCs that utilize the GPU, suitable configuration settings with respect to the available hardware resources. The configuration interface can only be connected to a constant. The interface is locked and cannot be connected to other data ports. The interface is a mandatory element when defining a GPU-aware component.

The connection between two data ports from different categories, such as an output data port and a GPU input data port, is done through an adapter which is automatically generated. The purpose of the adapter is to automatically transfer data from one memory system to another and to supply the right data type to the connected software circuits. For example, when a regular output data port is connected to an input GPU data port, a CPU-to-GPU adapter is generated to automatically shift the data from the RAM system to the GPU memory and to provide the appropriate data type with respect to the GPU memory architecture. There are two types of adapters: a CPU-to-GPU adapter which copies data from the RAM system onto the GPU memory system, and a GPU-to-CPU adapter which transfer the data onto the RAM system.

Adapting the vision system with our proposed solution, the new revised design is presented in Fig. 4. The system design is enriched with new input and output GPU data ports for the GPU-aware components *ImageMerger* and *ColorFilter*. New adapters automatically assist the data transfer operations (i.e., CPU-to-GPU and GPU-to-CPU) between the *ColorFilter* and *VisionManager* and *Logger*. The same applies between the camera components and *ImageMerger*. We decide to not graphically represent the adapters in the architectural view as the data shifting operations are performed in a transparent way. *ImageMerger* and *ColorFilter* GPU-aware components use the configuration interface to receive their specific computation settings.

The altered activities of the modified vision system design are illustrated in Fig. 5. The automatically introduced adapters are handling the data transfer between the CPU and GPU: two adapters are shifting the initial camera frames onto the GPU RAM, while a third one copies the result of the *ColorFilter* SWC back onto the CPU RAM system. Equipped with GPU ports, *ImageMerger* communicates directly with *ColorFilter* through the GPU RAM environment.

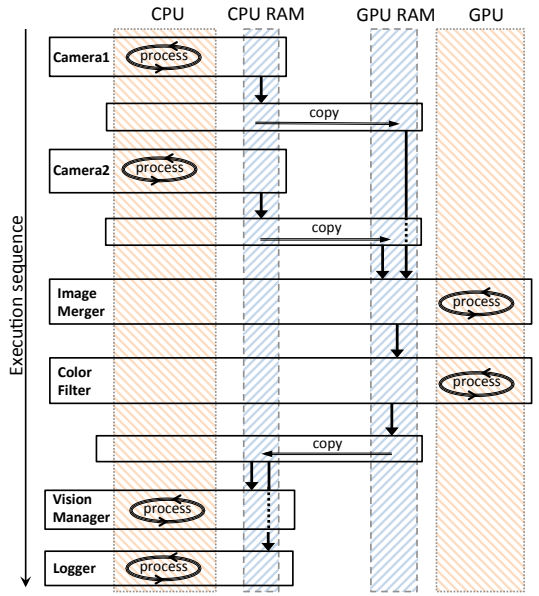


Fig. 5: The hardware activity of the revised vision system

When using the extended Rubus component model to develop GPU-based applications for embedded systems, the following advantages are found:

- The communication between GPU-aware SWCs is kept locally, onto the GPU RAM system. This decreases the communication overhead. Therefore, the system performance is improved and the stress over the hardware communication CPU-GPU bridge is reduced.
- As an effect of introducing the adapter communication elements, the code duplication in the GPU-aware SWCs is reduced. Another advantage of externalizing the data transfer operations through the adapters is that the GPU-aware component becomes lighter from the computational concern.
- The reusability of the SWCs is enhanced by distributing the computation settings at the system level. In this way, suitable computation settings with respect to the GPU computation resource are assigned to GPU-aware SWCs at system level.

V. Realization

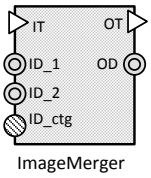
The extension includes new component model elements: *GPU data ports*, *configuration interface* and *adapters*. Using the current Rubus framework, we create the introduced concepts by using existing Rubus constructs. Their realizations are described in the following paragraphs.

A. Ports

The realization of the *GPU data ports* and the *configuration interface* is done by using regular Rubus data ports.

In the following paragraphs, we present how a GPU-aware component with GPU data ports and configuration interface is implemented using the Vision System running example. In Rubus, a software circuit is specified by a header and a source C file. The header file declares the structures used for the component interface and the behavior function, while the source file defines the behavior through the entry function. Fig. 6 shows the header C file of the *ImageMerger* software circuit.

A Rubus component is characterized by an interface that contains all its input and output data ports. The interface declaration of the *ImageMerger* component is *SWC_merger_iArgs_t*. It contains two structs: the *IP_SWC_iArgs_t* for the input data ports and the *OP_SWC_iArgs_t* for the output data ports. Using the existing port implementation constructs, we define the port of the configuration interface as a regular input data port. Therefore, the struct of the input data ports contains, besides the input data ports ID_1 and ID_2, the configuration settings port ID_cfg. This configuration port, declared as a struct, contains all the parameters required to specify how much of the GPU threads the component consumes at the run time. ID_1 and ID_2 are GPU data input ports but are realized as




```
typedef struct {
  GPU_unsigned_char *ptr;
  int width;
  int height;
}img_format;

//specific GPU settings
typedef struct {
  int blockDim_x, blockDim_y,
      blockDim_z;
  int gridDim_x, gridDim_y,
      gridDim_z;
}GPU_settings;

//GPU input ports and the
    configuration interface
typedef struct {
  img_format *ID_1;
  img_format *ID_2;
  GPU_settings *ID_cfg;
}IP_SWC_iArgs_t;

//GPU output port
typedef struct {
  SWC_img_format OD;
}OP_SWC_iArgs_t;

typedef struct {
  IP_SWC_iArgs_t IP;
  OP_SWC_iArgs_t *OP;
}SWC_merger_iArgs_t;

extern void SWC_merger_entry (
    SWC_merger_iArgs_t *args);
```

Fig. 6: Implementation details of the *ImageMerger* component

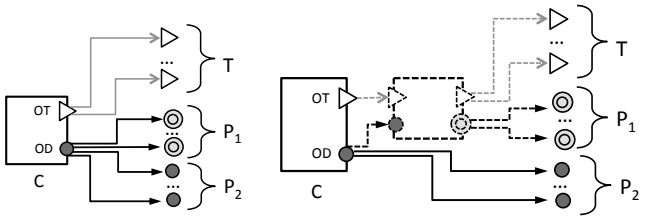
regular data ports. They are declared as structs that contain information about the image frame, such as width, height or the memory location. Similarly, the GPU output port, declared

as a struct that contains information about the merged image frame, is declared as an element of the output port struct *OP_SWC_iArgs_t*. The behavior function *SWC_merger_entry* is defined in the source C file of the component.

B. Adapters

Using the existing Rubus framework, the *adapter* is realized as a software circuit with GPU data ports and regular data ports. The CPU-to-GPU adapter is realized as a software circuit equipped with an input data port and one GPU data output port, while the GPU-to-CPU adapter is a software circuit with one input GPU port and one output data port.

Being realized in Rubus as software circuits, the adapters follow the ordinary Read-Execute-Write semantics. Whenever the trigger port of the adapter is activated, the data from the input data port are read in an atomic operation, followed by the switch of the internal state to the execution mode during when the functionality is performed. After the data transfer is finished, the SWC's internal state becomes inactive and the output port is atomically written. Finally, the trigger output port is activated.

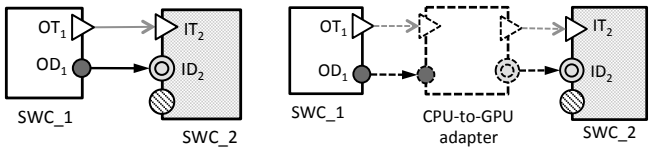


(a) Rubus extension (b) Realization of the adapter

Fig. 7: The general case of the CPU-to-GPU adapter realization

A CPU-to-GPU adapter is generated when an output port of a software circuit *C* is connected to at least one GPU-aware software circuit. Consider a software circuit *C* with an output trigger port *OT* and an output data port *OD*. Let *T* be a set containing all trigger ports that are connected to *OT* and *P* the set composed of all data ports that are connected to *OD*. For the general case, the set *P* is constructed from two subsets, P_1 which contains only GPU data ports and P_2 consisting of data ports, as described in Fig. 7(a). The connection realizations between the component *C*, a CPU-to-GPU adapter and all the components that are connected with *C*, are done as illustrated in Fig. 7(b). The *OT* and the *OD* ports are connected to the input trigger port and the input data port of the adapter, respectively. The output trigger port of the adapter is connected to all ports from the *T* set. The GPU output data port of the adapter is only connected to the ports from the subset P_1. With the generation of the adapter, all the connections between the *OT* port and the ports from the *T* set and the port *OD* with the ports from P_1 set respectively are removed. Similar rules apply when a GPU-to-CPU adapter is automatically generated.

To exemplify the connection realizations of a CPU-to-GPU adapter, we describe two examples. Fig. 8(a) presents

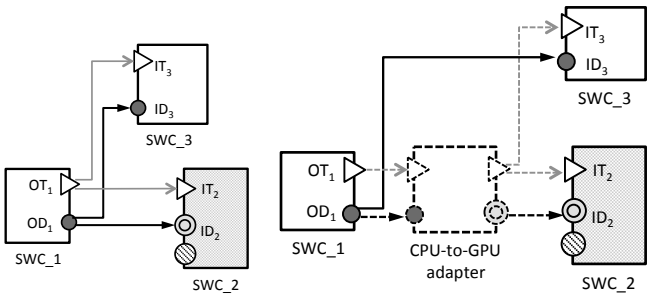


(a) Rubus extension (b) Realization of the adapter

Fig. 8: A simple example of adapter realization

the simplest case, where a software circuit *SWC_1* is only connected to a single GPU-aware software circuit *SWC_2*. A CPU-to-GPU adapter is generated and placed between *SWC_1* and *SWC_2*, as shown in Fig. 8(b).

A more complex situation is depicted in Fig. 9(a), where a component *SWC_1* is connected to a GPU-aware component *SWC_2* and to a regular component *SWC_3*. The connection realizations between the adapter and all three components are realized similarly as described in the previous example, where the OT_1 and OD_1 are connected to the trigger and data ports of the adapter. Both *SWC_2* and *SWC_3* components are triggered by the adapter, by connecting the output adapter trigger port to IT_2 and IT_3. Copying the data onto the GPU RAM system, the adapter output port connects to only ID_2 port. The initial connection between the OD_1 and ID_3 regular data ports is not replaced by the generation of the adapter, as described by the Fig. 9(b).



(a) Rubus extension (b) Realization of the adapter

Fig. 9: A complex example of adapter realization

To depict the utilization of the proposed Rubus adapters, we present the implementation adapter details from the system illustrated in Fig. 10. In the Vision System, three adapters are generated: two CPU-to-GPU adapters copy images onto the GPU memory for the *ImageMerger* component, and one GPU-to-CPU adapter copies the resulted filtered images back onto the CPU RAM for the *VisionManager* and *Logger* components.

Each adapter, implemented as a software circuit has a header and a behavior file. An adapter header is implemented similar to the one described in Fig. 6. The behavior of a CPU-to-GPU adapter is presented in Fig. 11. The adapter allocates the right amount of memory onto the GPU using the *cudaMalloc* specific function, where it shifts the data from the main memory system through the *cudaMemcpy* operation. The data from the main memory is represented by a pointer variable

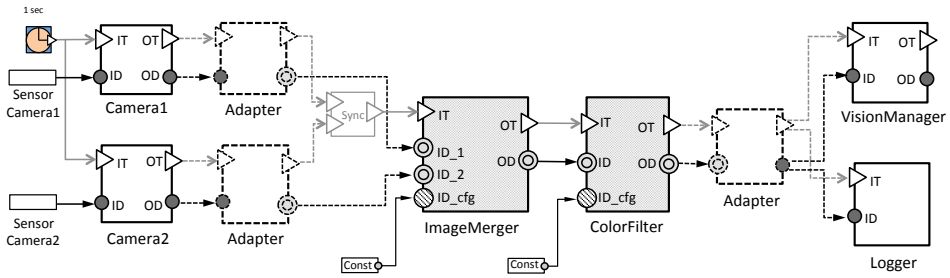


Fig. 10: The vision system design and the automatically generated adapters

host_ptr which points to the memory location of the data that needs to be shifted. Similarly, the *device_ptr* variable points to the memory location from the GPU system, where the data is shifted. After the copy operation is finished, the adapter sets its GPU output port with the necessary values.

```
void adapter_entry (adapter_iArgs_t *args)
{
      GPU_unsigned_char *device_ptr;
      unsigned_char *host_ptr;
      int width, height;
      width = args->IP.ID_input->width;
      height = args->IP.ID_input->height;

      cudaMalloc(&device_ptr, 3*sizeof(device_ptr) *
           width * height );
      cudaMemcpy(device_ptr, host_ptr, 3*sizeof(
          host_ptr)* width * height,
          cudaMemcpyHostToDevice );

      IPA_OP_System__adapter = {{device_ptr, width,
          height}};
}
```

Fig. 11: The behavior of a CPU-to-GPU adapter

VI. Evaluation

By extending the Rubus component model with our solution, on one hand, we increase the overhead of the system due to the execution of additionally introduced software circuits (i.e., adapters). On the other hand, we decrease the communication overhead using the same introduced component elements. In this section, we present two experiments to investigate the efficiency of the proposed extension. The experiments are performed on a system containing an Unix-based OS (i.e., Ubuntu 15.04LTS), a hardware platform consisting of an NVIDIA Quadro 600 GPU hardware with Fermi architecture and an Intel Xeon i7 CPU with 32 GB of internal RAM.

A. Experiment 1

In the first experiment, we compare the end-to-end execution time of two versions of the vision system. The first version is built with the proposed extension while the second version is built with the standard Rubus solution that encapsulates all GPU information within ordinary SWCs, in both versions. The end-to-end times are counted from the triggering of the first component of the system (i.e., *Camera1*) until the last triggered component (i.e., *VisionManager*) finishes its execution. The experiment investigates the overhead caused by the extension when varying the size of the system input data. We use three different input sets that contain two images respectively of 512 x 512 pixels, 1024 x 1024 pixels and 2048 x 2048 pixels. The same GPU computation settings are used for both versions of the system. We execute each system version for 1000 times and for each time, the end-to-end times are measured. Based on this, an average value of the end-to-end times is calculated.

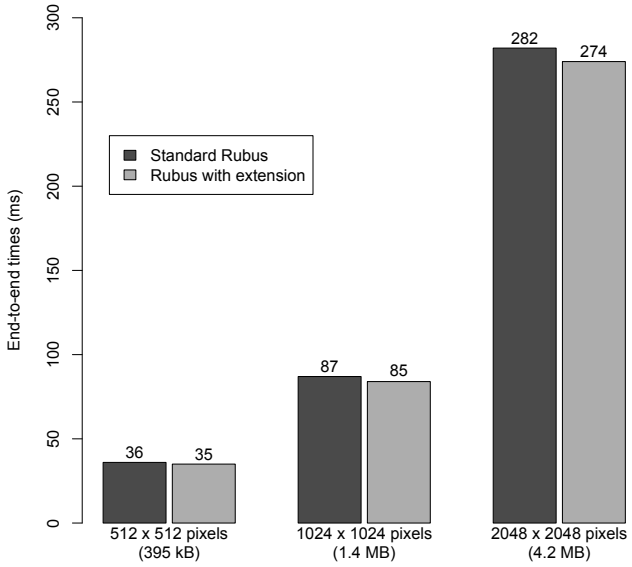


Fig. 12: End-to-end times in the vision system while processing different input sizes

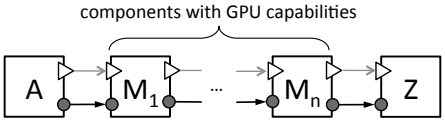
Fig. 12 presents the measurements of the two versions of the vision system, for inputs of different sizes. Despite having additionally generated components (i.e., adapters), the performance of the extended Rubus is comparable to the standard solution. This is as expected as the system under study is

relatively small, only containing two GPU-aware components connected in a sequence. However, when increasing the size of the input, we can notice an increased efficiency when using our proposed extension. In each of the six experiment cases, less than 2% of the measurements deviate more than $\pm 0.1\%$ from the calculated average value.

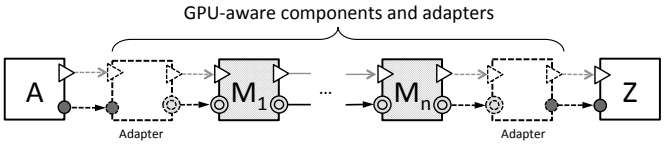
Regarding the number of code lines provided by the user for the two components with GPU capabilities (i.e., *ImageMerger* and *ColorFilter*) there is approximatively 30% less code (25 code lines) when using the extended Rubus solution, due to the ability to automatically take care of GPU specific copy operations.

B. Experiment 2

The second experiment investigates the produced overhead when utilizing the extension, while increasing the number of components with GPU capabilities. For this experiment, we construct two versions of a component, both using the GPU to execute the same functionality, i.e., the mirroring of an image. One version is a regular Rubus component with data ports, encapsulating the necessary GPU information and operations, and the other is build as a GPU-aware component based on our proposed extension. Based on them, we developed a system for each version: a system composed of regular software circuits using a standard Rubus solution and a system with adapters and components equipped with GPU data ports. To keep it simple, each system is a chain of sequentially connected instances of the corresponding mirror component version.



(a) A system based on a standard Rubus solution



(b) A system based on a Rubus with extension solution

Fig. 13: The systems used in Experiment 2

Fig. 13 illustrates the two mirroring systems used in this experiment. A system built with a standard Rubus solution is presented in Fig. 13(a), where an initial component *A* examines the image to be mirrored for e.g., the number of pixels to be processed, and an ending component *Z* uses the mirrored final output. Fig. 13(b) illustrates the second mirrored system based on a Rubus with extension solution, composed of GPU-aware components and adapters. We modify the number of components with GPU capabilities in order to evaluate the changing overhead of the system with extension. For both mirror systems we use the same GPU computation settings and an image of 2048 x 2048 pixels (4.2 MB) to be mirrored. We execute each system for 1000 times and calculate the average of the measured end-to-end time.

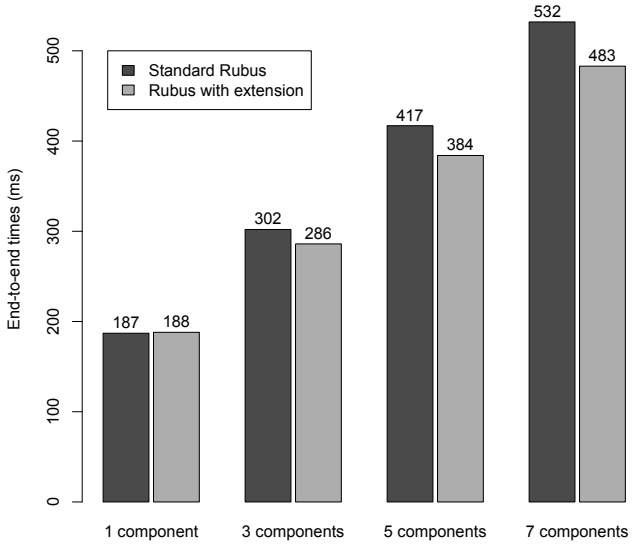


Fig. 14: End-to-end times in the system when varying the number of composed components

Fig. 14 presents the results of both systems when varying the number of components. When the systems contain a single mirror component, the system based on a standard Rubus solution is performing relatively the same as our proposed extension. Hence, the overhead introduced by the additional generated components is negligible. When the systems are composed from three to seven components, the gain of our proposed extension is gradually increasing from 6% to 10%. Also in this experiment, less than 2% of the measurements deviate more than $\pm 0.1\%$ from the calculated average value, in each of the eight cases. This experiment shows that the extended Rubus solution performs better than the standard solution, when the software architecture is composed of more than one GPU-aware component.

The number of user provided code lines for the mirror component is decreased with 25% when using the extended Rubus solution compared to the standard approach.

VII. Related Work

The solution of using heterogeneous hardware to tackle the highly demanding resources is already embraced by the industry. For example, multi-core ECU platforms are adopted in the aeronautics and space industry [13], where the Integrated Modular Avionics architecture [14] is extended to support the new hardware platforms. Another example is the AUTOSAR component model from the automotive industry which has

been extended to support multi-core ECUs [15]. A step forward in the development of heterogeneous embedded systems is achieved through hybrid system-on-chip that contain both CPU and field-programmable gate array (FPGA). Andrews et al. describe a way to use COTS components, referred as hybrid components, to address the CPU-FPGA hardware [16]. The work started the development of a unified programming model based on the multi-threading programming paradigm to synchronize CPU-FPGA computations within the hybrid components [17]. Although the GPU-aware components proposed by our Rubus extension use the CPU to trigger the execution on the GPU, these GPU-aware components are not similar to the hybrid components from the computation perspective. In our solution, the communication between components with CPU and GPU computations is done through the component interface, assuring a high degree of reusability for the components.

Regarding CPU-GPU systems, the PEPPHER component-model presents a way to efficiently utilize heterogeneous CPU-GPU hardware [18]. The component model uses the PEPPHER component which is an annotated software unit with an interface. The interface is defined by an XML descriptor document that specifies the name, parameter types and access type of the component. There may be several implementation variants of the same functionality (e.g., for CPU and GPU) defined by the component interface. The parameters passed between PEPPHER components are wrapped in special portable and generic data structures. These structures, called smart containers, ensure the memory transfer of data between processing units and do the memory management of the copied data. In our work, we provide a transparent and automatic way of transferring data between processing units by using adapters. Similar to the smart containers which are more complex when dealing with memory management, our adapters can be considered as high-level memory management elements.

The Elastic computing [19] is a component-based framework that statically determines the optimal software configuration for a given platform using a library that contains prebuilt ”elastic functions”. A limitation of this work is that the resource allocation and data management is done inside the elastic function, which we tackle in our proposed extension through the adapters.

It is worth mentioning that there is a lot of work addressing development of models for heterogeneous hardware. Several programming models target systems with multi-core CPUs and accelerator units (e.g., GPUs). They can be split into two categories, direct and library-based approaches. Example of direct approaches include the work of Papakipos [20] that provides a C and C++ API. The API calls are dynamically translated into parallel programs which are executed across multiple processing units. The Merge framework [21], which is a library-based approach, automatically and dynamically parallelizes and load-balances the application computation over the available hardware processing cores (e.g., CPU, GPU). The work unit is a task and the application computation is composed from a number of tasks which are mapped to special functions that encapsulate the accelerator-specific code.

Numcode

VIII. Conclusion

The latest technology progress favored the development of heterogeneous embedded systems. Becoming more and more popular, the embedded systems that contain CPUs and GPUs represent a viable solution for applications that are highly resource demanding. Despite the latest growing interest in CPU-GPU embedded systems, the existing component models do not provide any support for GPU usage. Thus, all the specific information and operations required to access and use the GPU are encapsulated in the components. This negatively affects the component reusability and the communication between components. In this paper, we have extended the Rubus component model to provide support for GPU development. We have introduced new Rubus artifacts and presented their specifications and semantics using a running example. The benefits of our Rubus extension include:

- The system performance is increased by reducing the overhead communication between SWCs.
- The duplicated code of SWCs is reduced by adding specialized communication elements.
- The reusability of SWCs is improved by deciding the GPU configuration settings at the system level.
- It is worth nothing that our solution is not limited to Rubus and is applicable to any component model that has similar characteristics, in particular the pipe & filter interaction style (e.g., SaveCCM [22], ProCom [23]).

For future work, we propose to increase the efficiency of the extended Rubus solution by allowing parallel execution of GPU-aware components. When a GPU-aware component is executed on the GPU and is not consuming all the hardware resources (e.g., GPU memory, computation threads), one or several GPU-aware components may be executed on the same GPU hardware, if possible. In this way, the execution time of the system is improved and the hardware is fully utilized.

Acknowledgments

Our research is supported by the RALF3 project[1] - (IIS11-0060) and the PRESS project through the Swedish Foundation for Strategic Research (SSF).

References

[1] P. Michel *et al.*, “GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing,” in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, 2007, pp. 463–469.

[2] D. Geronimo, A. M. Lopez, A. D. Sappa, and T. Graf, “Survey of pedestrian detection for advanced driver assistance systems,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 7, pp. 1239–1258, 2010.

[3] “NVIDIA CUDA C programming guide,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: 2015-12-02.

[4] “The OpenCL specification version 2.1,” https://www.khronos.org/developers/reference-cards/, accessed: 2015-12-02.

[5] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002.

[1] http://www.mrtc.mdh.se/projects/ralf3/

[6] T. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge," in *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, 2006, pp. 1–15.

[7] T. Bures, P. Hnetynka, and F. Plasi, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*. IEEE, 2006, pp. 40–48.

[8] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[9] "AUTOSAR - Technical Overview," http://www.autosar.org, accessed: 2015-12-02.

[10] 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, pp. 177–183.

[11] G. Campeanu, J. Carlson, and S. Sentilles, "A GPU-aware component model extension for heterogeneous embedded systems," in *The Tenth International Conference on Software Engineering Advances*, November 2015. [Online]. Available: http://www.es.mdh.se/publications/4088-

[12] C. Ahlberg, L. Asplund, G. Campeanu, F. Ciccozzi, F. Ekstrand, M. Ekstrom, J. Feljan, A. Gustavsson, S. Sentilles, I. Svogor *et al.*, "The Black Pearl: An autonomous underwater vehicle," 2013.

[13] R. Fuchsen, "How to address certification for multi-core based IMA platforms: Current status and potential solutions," in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*. IEEE, 2010, pp. 5–E.

[14] P. J. Prisaznuk, "Integrated modular avionics," in *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*. IEEE, 1992, pp. 39–45.

[15] F. Kluge, M. Gerdes, and T. Ungerer, "AUTOSAR OS on a message-passing multicore processor." in *SIES*, 2012, pp. 287–290.

[16] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid FPGA-CPU computational components: a missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[17] D. Andrews, D. Niehaus, and P. Ashenden, "Programming models for hybrid CPU/FPGA chips," *Computer*, vol. 37, no. 1, pp. 118–120, 2004.

[18] S. Benkner, S. Pllana, J. L. Traf, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "Peppher: Efficient and productive usage of hybrid computing systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, 2011.

[19] J. R. Wernsing and G. Stitt, "Elastic computing: A portable optimization framework for hybrid computers," *Parallel Computing*, vol. 38, no. 8, pp. 438–464, 2012.

[20] M. Papakipos, "The PeakStream platform: High-Productivity software development for multi-core processors," *Writepaper, PeakStream Corp*, 2007.

[21] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, no. 2. ACM, 2008, pp. 287–296.

[22] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "SaveCCM - a component model for safety-critical real-time systems," in *Euromicro Conference, 2004. Proceedings. 30th*. IEEE, 2004, pp. 627–635.

[23] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A component model for control-intensive distributed embedded systems," in *Component-Based Software Engineering*. Springer, 2008, pp. 310–317.