Flexible Components for Development of Embedded Systems with GPUs

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

***Abstract*—Today, embedded systems incorporate GPUs through a multitude of different architectures. When it comes to the development of these systems with GPUs, component-based development is ill-equipped as it does not provide support for GPUs. Instead, the component developer needs to encapsulate inside the component, besides functionality, settings and environment information that are specific to a particular GPU architecture. This binds the component this GPU architecture. Using these hardware-specific components characterized by restricted reusability, the system developer is confined to a limited design-space which may negatively impact the overall system feasibility.**

In this paper, we introduce the concept of flexible components, which are components that can be executed indifferently on CPU or GPU, regardless of the architecture. Using flexible components, component developers are relieved from the side development activities (e.g., environment information) which are automatically handled by component-level mechanisms. To enhance component communications, connection elements (i.e., adapters) are generated to handle component data transmission, taking in consideration the platform characteristics. Finally, our proposed solution is evaluated by using flexible components to implement the vision system of an underwater robot, and execute it on three platforms with different GPU architectures.

I. Introduction

Many modern embedded systems deal with huge amounts of data originating from the interaction with the environment. For example, the autonomous car developed by Google[1] processes up to 750 MB of data per second delivered through its sensors (e.g., cameras). The data must be processed with a certain performance in order to handle, in real-time[2], the environment changes. For example, an underwater robot must rapidly process environment data in order to identify moving objects before they actually move away from the sensors' range.

A solution to process these data with adequate performance is the usage of general-purpose Graphics Processing Units (GPUs), which, thanks to their architecture, excel for highly data-parallel applications. Today, embedded-board platforms contain GPUs and different platforms often have different architectures. Depending on their characteristics (e.g., size, energy consumption, computation power), different platforms are suitable in different contexts. For example, there are platforms with high-computation GPU such as Condor GR2 used in high-performance computing solutions, but also low-computation with low energy consumption such as Mali-470 GPU, utilized in smart watches.

Another trend in the development of embedded systems is the usage of Component-Based Development (CBD) [1]. This software engineering methodology promotes the development of systems through the composition of already existing software units called (software) components. The industry successfully uses various component models such as AUTOSAR [2], Rubus [3] and IEC 61131 [4].

However, CBD is ineffective for embedded platforms that combine CPUs and GPUs. This is due to the lack of specific support for GPUs. This overall challenge has several facets. One of them refers to the development of components with GPU capabilities, which is complex, time-consuming and error-prone. The component developer effort is increased due to the fact that alongside the functionality, the component must contains settings and GPU-specific environment information. Therefore, encapsulating inside the components all the required information, results in hardware-specific components suitable for particular GPU architectures only. In our previous work [5][6], we tackled parts of this issue in introducing: *i)* specialized components that are specific designed to encapsulate GPU functionality and cannot function without a GPU hardware, and *ii)* particular artifacts (i.e., adapters) that facilitate component communications by automatically transferring data between CPU and GPU memory systems.

Another existing issue involves the reduced flexibility of the current way in designing component-based applications with GPU capabilities. The existing hardware-specific components have a reduced reusability between different hardware contexts. Using these (hardware-specific) components makes the system developer to be limited, to some extent, in exploring all the design-space solutions.

In this work, we introduce flexible components. Basically, a flexible component is a component with a functionality that may be executed either on CPU or GPU. Using flexible components, the component developer focuses only on implementing the functionality. Component-level mechanisms automatically generate environment-specific information that allows the component to be executed on different hardware. Moreover, when employing flexible components in the system design, the

[1] https://waymo.com/

[2] Real-time refers here to the timing correctness aspect, i.e., the physical time when the system's results are produced

system developer has a larger design-space to choose from. In the context of flexible components, the adapters automatically transfer data between components, taking in consideration the platform specifications.

The benefits of employing flexible components are: *i)* automatized mechanisms lift from the component developer the responsibility of handling the component environment-specific information, *ii)* a larger design-space may positively impact the overall system feasibility, and *iii)* generated adapters improve the component communication efficiency.

The reminder of the work is divided as follows. Background of GPUs and CBD in embedded systems is covered by Section II followed by the solution overview in Section III. We present the realization details of the solution in Section IV. The feasibility evaluation is covered by Section V, followed by related work (Section VI) and conclusions (Section VII).

II. Background - CBD and GPUs

In the first part of this section, we presents more details about CBD and its use in the context of embedded systems. The second part describes the GPUs, their characteristics and how they are addressed by the existing programming models.

A. CBD in Embedded Systems

Component-based development is a software engineering methodology that promotes the efficient system development through the composition of already existing software blocks called (software) *components*. CBD advertises the use and reuse of the same component in different contexts, which increases the development efficiency. The way components interact is through *interfaces* which are specifications of the components' access points. Several types of interfaces exists, such as *port-based* and *operation-based* interfaces. The port-based interfaces, used in our work, comprise of access points for sending/received data of different types between components.

A key concept of CBD is *encapsulation*, where all component information including implementation details, are enclosed inside the component. There are four level of the encapsulation. Black-box components are in compiled or binary form, that cannot be changed; their functionality can be accessed through the interface. The white-box components, used in this work, are readable source code, directly changeable by the programmers. The glass-box components' functionality is accessed through the interface, and their internals are visible from outside. For gray-box components, the developer has access to their interface and internals.

A component is constructed by following the specifications of a *component model*. Besides the construction rules, the component model defines how components interact with each other when they are assembled into a system [7].

The embedded and real-time system industry successfully adopted the CBD methodology through various component models. These component models follow different interaction styles suitable for particular type of applications [8]. For example, AUTOSAR which is a standard in automotive industry, follows the *request-response* and *sender-receiver* interaction styles. The *pipe-and-filter* style used in our work, is another interaction approach used by industrial component models such as IEC 61131 and Rubus. This particular style, suitable for streaming of events-type of applications, allows an easy mapping between the interaction model and the control specifications required by embedded and real-time systems.

As the evaluation uses the Rubus component model, we provide more information about it. From the development perspective, a Rubus component consists of three parts, i.e., a constructor, behavior function and destructor. The constructor is executed once, before the system execution, and allocates the component resource requirements, while the destructor is execution when the system is properly switched off, and releases the allocated resources. The behavior function contains the functionality of the component and is executed each time the component is triggered.

B. GPUs

GPUs were developed back in 90s and were employed only in graphic-based applications. In time, thanks to the increase of their computation power and becoming easier to program, GPUs were utilized in different type of applications becoming general-purpose GPUs [9]. For example, cryptography applications [10] and Monte Carlo simulations [11] have GPU-based solutions.

The GPU is a processing unit equipped with a parallel architecture that may employ at a time, thousands of computation threads through its multiple cores. Due to its execution model, GPU excel in executing data-parallel applications. For example, a simulation of bio-molecular systems may achieve 20 times speed-up on GPU compared to the CPU [12].

A particularity of a GPU is that it cannot function without a CPU. Considered as the brain of the system, the CPU coordinates all the GPU-specific activities such as data transfer or execution of GPU functionality. Embedded-board platforms with different GPU architectures are now developed by various vendors. There are two existing types of GPU architectures, where:

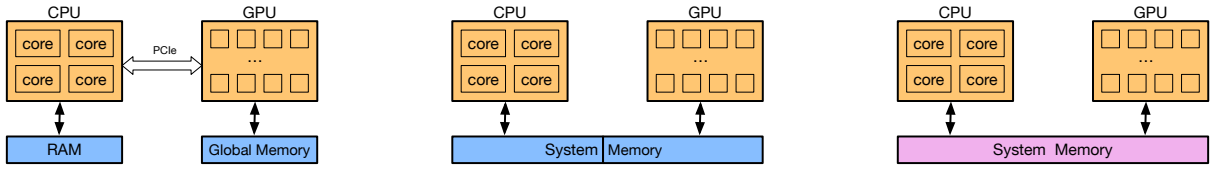
- the GPU is discrete (*dGPU*) and has its own private memory such as the Condor GR2[3]; and
- the GPU is integrated (*iGPU*) on the same chip as the CPU (referred as SoC), sharing the same memory, such as AMD Kabini[4].

Embedded-boards with iGPU architectures are the predominant platforms used in industry due to their reduced cost, size and energy usage. On the other side, dGPUs, with large physical sizes that incorporates more (GPU) resources, are used by systems that require higher performance.

Fig. 1 presents different platforms with GPU architectures, and their memory characteristics. Systems with dGPUs (Fig. 1(a)) have different memory systems and data needs to be

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

[4]http://www.amd.com/en-us/products/processors/desktop/athlon



(a) System with discrete GPU (b) SoC with distinct memories (c) SoC with shared memory

Fig. 1: Embedded platforms with different GPU architectures

copied (with additional overhead) via e.g., the PCIexpress bus. Platforms with iGPUs have physical memories that are divided into distinct parts (Fig. 1(b)). In this case, there is still need for data transfer activities but the copy overhead is minimized due to the physical location (i.e., on the same memory chip). For this type of architecture (i.e. iGPU), there are platforms that optimize the memory access by offering a shared virtual memory (SVM) space. To place data on SVM, specific transfer activity are used; on the other hand, no specific activities are used to access the data from SVM. The latest SoC architecture (Fig. 1(c)) offers concurrently access to the same memory for both CPU and GPU, without the need for data copies.

Regarding the development of applications with GPU capabilities, different programming models may be employed. OpenCL[5] is a general framework that is supported by multiple platforms and vendors (e.g., Intel, AMD, NVIDIA, Altera, IBM, Samsung, Xilinx). In this work, we utilize OpenCL to construct our flexible component solution.

While using OpenCL to develop an application, there are several hierarchical steps that needs to be respected. A *platform* is at the very top level; it contains the installed vendor's driver. A platform needs to have its own *context* that may contain one or several execution *devices*. For example, a system may have three devices, i.e., one CPU and two GPU (iGPU and dGPU) devices. A device should be selected in order to execute the functionality. The commands given by the host (i.e., CPU) to the device (e.g., iGPU) are sent using a *command queue* mechanism. The functionality that is intended to be executed on the device, also known as the *kernel*, may be defined even before setting the platform. A next step is the allocation of device memory to hold data (referred as *buffers*), either as input or output for the kernel function. A *program* to hold the defined kernel is created and compiled. The kernel arguments are assigned by using the allocated (input and output) buffers, and specify the number of threads utilized for the kernel execution. Finally, the kernel is ready to be executed, and its results are transfered back to the host. As a last step, the resources (i.e., memory buffers, program, context, command queue, kernel) are released.

III. Flexible development of embedded systems with GPUs

In this work, we introduce the notion of *flexible component* which has the following particularities. A flexible component is a white-box component, with readable and modifiable source code. The functionality of a flexible component is expressed in a parallel manner using the OpenCL syntax. This functionality can be executed either on CPU or GPU and the component does not contain any environment-specific information that would bind it to a particular processing unit.

During system design, the system developer decides on which hardware (i.e., CPU or GPU) the flexible components should be allocated onto. In order to be executed on the specified hardware, the required environment information is generated automatically.

The benefits of adopting flexible components are the following:

- The component developer does not need to address the particular environment settings required for the component execution, including details such as allocating GPU memory and transferring parameters to the GPU; hence, the development effort and complexity is decreased.
- The system developer has more alternatives when exploring the system design-space solutions, which may result in an improved overall system feasibility.
- Efficient communication between components allocated on different parts (i.e., CPU and GPU) of the platform can be automatically generated.

A. Flexible component

We define a flexible component C_{flex} as a tuple that contains the functionality F disclosed through the interface I. Flexible components use port-base interaction style and the underlying component model uses a pipe-and-filter style.

$$C_{\text{flex}} = \langle F, I \rangle, \textit{ with } I = \{p_1, p_2, ...\}.$$

F denotes the functionality.
I defines a set that contains all data ports p_k.

Through the ports of the interface, C_{flex} 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:

$\forall p_k \in I$, $p_k.name$ denotes the name of port p_k, and
$p_k.type$ denotes the type of port p_k.

The ports of flexible components support data of regular type such as *integer* or *double*. Besides regular data, we

[5]https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/

introduce *m-elem* as a data type, referred as multi-element type, to describe large data that is also supported by flexible component ports. For example, a 2D image is a large data composed of many pixel elements grouped together, that would be represented by a multi-element type.

In order to generate all the parts required by a flexible component to be executed (see Section IV), we define additional port information related to the multi-element ports, as follows.

The data of the multi-element type may have several dimensions. Moreover, data of m-elem type contain many elements, and the size of the individual elements can vary. Therefore:

$\forall p_k$, where $p_k.type =$ *m-elem*

$p_k.width$ denotes the horizontal size,

$p_k.height$ denotes the vertical size, and

$p_k.size$ denotes the size of each data element.

A good example is a color (2D) image with two dimensions, that is a width of 640 and a height of 480 pixel elements. Each pixel of the image has a color (i.e., value) that is obtained from a combination of three colors (Red, Green and Blue), each with values between 0 to 255. The size of each color pixel is $3 * 1$ bytes (the *unsigned char* type allows values between 0 and 255 and has a size of 1 byte). For a multi-element data with one dimension (e.g., array of integers), $p_k.height$ is set to 1. The approach can easily be extended to support data with more than two dimensions, e.g. by introducing information about *depth* for 3D images.

The ports of the interface I are grouped into two subsets, i.e., input and output. Moreover, each subset is further divided according to their data types (i.e., regular and multi-element). Therefore:

$$I = I_{\text{in}} \cup I_{\text{out}}, \text{ where}$$
$$I_{\text{in}} = I_{\text{reg\_in}} \cup I_{\text{multi\_in}} \text{ and}$$
$$I_{\text{out}} = I_{\text{reg\_out}} \cup I_{\text{multi\_out}} \text{ with}$$

$$\forall p_k \in I_{\text{reg\_in}} \cup I_{\text{reg\_out}},\ p_k.type \neq \textit{m-elem} \text{ and}$$
$$\forall p_k \in I_{\text{multi\_in}} \cup I_{\text{multi\_out}},\ p_k.type = \textit{m-elem}.$$

As described in the background section, 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. The motivation of introducing the multi-element type and the separation of the interface into two subsets is that automatically generated mechanisms transfer data between CPU and GPU based on the data type information of the connected ports. For example, for platforms with dGPUs, when a regular component sends a data of m-elem type to a GPU-allocated flexible component, that specific data is automatically transfered on the GPU memory. We do not need to specifically handle data of a regular type (e.g., integer or double) because the GPU run-time driver automatically handles it.

The other part of the flexible component is its functionality F. The flexible component functionality is basically the kernel code (see Section II) constructed using the OpenCL syntax. Written in a data-parallel manner, it can be executed by either CPU or GPU.

From the implementation perspective, the multi-element type contains a *pointer* to the memory location of the data. Listing 1 outlines the construction of the multi-element type.

Listing 1: Declaration of the *m-elem* type

```
typedef struct{
      unsigned char *data;
      } m-elem;
```

Concerning the actual implementation of the functionality, instead of hard-coding the m-elem port information (e.g., 640) inside the functionality which is considered a bad practice (e.g., poor modifiability, readability), we provide a number of macros. Moreover, we hide through a macro definition the fact that a buffer (with one value) is utilized when addressing regular output data. The macro definitions located in the flexible component constructor (see Section IV-A), are the following:

$\forall p_k$, where $p_k \in I_{\text{multi\_in}} \cup I_{\text{multi\_out}}$

$p_k.name\_width$ to access the width information,

$p_k.name\_height$ to access the height information,

$p_k.name\_size$ to access the size information, and

$p_k.name\_data$ to access the data memory location.

$\forall p_k$, where $p_k \in I_{\text{reg\_out}}$

$p_k.name$ to access the corresponding (one value) buffer.

IV. Realization

The realization of our solution is implemented on two levels, i.e., on the component and system level, as follows. In order for a flexible component to be executed on a device (i.e., CPU or GPU), it needs to include, alongside the component functionality, specific information such as settings, environment details about the device and memory allocation to hold the result. As the flexible component contains the main functionality, component-level mechanism automatically generate the rest of the required code that allows it to be executed on the selected hardware. More specifically, using the core functionality and the information regarding the number and data types of the (input and output) data ports provided by the flexible component, a full component is generated, ready to be executed on the hardware. The resulting generated component contains: *i)* a *constructor* that initializes the component resource requirements, *ii)* a *behavior function* that represents the component functionality, and *iii)* a *destructor* that releases the component resources.

From the system level perspective, we introduce specific artifacts to facilitate an efficient component communication. More specifically, based on the component connections and component-to-hardware allocation, artifacts, referred as *adapters*, are generated where needed. The adapters take data from one component and provide it to the connected component in the appropriate memory location. For example, when

a regular component sends data to a GPU-allocated flexible component, the generated adapter automatically transfers data from the (CPU) main memory to the appropriate GPU memory location, corresponding to the platform characteristics.

More details about the component generation and communication are presented in the following subsections.

A. Component-level realization

This section describes the transformation of a flexible component into an executable component through its constituent parts, i.e., constructor, behavior function and destructor.

The *constructor* is a function that encloses all information necessary to initialize a component. The constructor contains: *i)* memory allocation to hold the component results, *ii)* the kernel functionality defined by the flexible component, and *iii)* settings and environment information of the hardware (i.e., CPU or GPU). The implementation provided by the developer in the flexible component represents the actual body of the kernel function. The kernel function name and its arguments are automatically generated inside the constructor using the information of the input and output data ports of the flexible component. This kernel function (name and its body) is loaded on the allocated hardware (i.e., CPU or GPU) as a string (due to OpenCL specifications).

Listing 2 describes the body of generated constructor function. At lines 3 and 7, the constructor contains the generated memory buffers that hold the resulting data corresponding to each output data port. We distinguish between regular and multi-element output ports. The OpenCL run-time does not support individual values (e.g., of integer or float type) as output kernel arguments. Therefore, when having an output data port of regular type, a buffer (that has one value) is created as an output argument of the kernel.

The generated constructor continues by defining a variable (i.e., *source_string*) that contains the kernel function name (line 10) and its arguments (line 12, 15, 18 and 21) that correspond to the input and output ports. The macro definitions described at lines 26-37 may be used by the component developer to access, inside the functionality, the required port information. The flexible component functionality is added to the string variable as the body of the kernel function (line 41). The device is loaded with the source code by storing the *source_string* variable into the program (line 45) after which is build (line 48). Finally, a kernel object (line 51) is created to be executed by the device.

Regarding the flexible component settings, we define two types. The first one refers to the device environment in which the flexible component will be executed i.e., the context, command queue and device id (at lines 3, 7, 45, 48). These settings are automatically constructed with the system initialization. If flexible components are allocated to both CPU and GPU, two global settings are constructed, each for the individual processing device. The same settings that target the GPU will be used for all GPU-allocated flexible components; similarly, CPU settings will be used for all CPU-allocated flexible components.

The second setting type concerns the specification and grouping of the computation resources of the hardware that are assigned to the component (lines 54 and 55). These settings are personalized to each flexible component and depend on e.g., the hardware available resources. Re-using the concept from our previous work [5][6], we provide these settings to each flexible component through a *configuration interface*. This interface is implemented as an additional input data port, that receives constant content and re-use it for every component execution.

Finally, the output m-elem ports contain the data resulted from the execution of the kernel by linking them to the corresponding memory buffers (line 59).

Listing 2: Constructor code

```
1  //create memory buffers for the output ports
2  <foreach p in I_reg_out>
3    void *result_<p.name> = apiCreateBuffer(settings->contex,
         CL_MEM_WRITE_ONLY, sizeof(<p.type>),NULL,NULL);
4  <endforeach>
5
6  <foreach p in I_multi_out>
7    void *result_<p.name> = apiCreateBuffer(settings->contex,
         CL_MEM_WRITE_ONLY, <p.width*p.height*p.size>,NULL,NULL);
8  <endforeach>
9
10  const char *source_string ="__kernel void flexible_kernel(
11 <foreach p in I_reg_in>
12   <p.type> <p.name>,
13 <endforeach>
14 <foreach p in I_reg_out>
15  __global <p.type> *result_<p.name>,
16 <endforeach>
17 <foreach p in I_multi_in>
18   __global <p.type> *<p.name>,
19 <endforeach>
20 <foreach p in I_multi_out>
21   __global unsigned char *result_<p.name>,
22 <endforeach>
23 ){
24 /* macro definitions to access port information */
25 <foreach p in I_multi_in ∪ I_multi_out>
26   #define <p.name>_width <p.width>
27   #define <p.name>_height <p.height>
28   #define <p.name>_size <p.size>
29 <endforeach>
30 <foreach p in I_multi_in>
31   #define <p.name>_data <p.name>->data
32 <endforeach>
33 <foreach p in I_multi_out>
34   #define <p.name>_data result_<p.name>
35 <endforeach>
36 <foreach p in I_reg_out>
37   #define <p.name> result_<p.name>[0]
38 <endforeach>
39
40 /* flexible component functionality */
41 <F>
42 }";
43
44 /* Create a program from the kernel source */
45   cl_program program = clCreateProgramWithSource(settings->context
         , 1, (const char **)&source_string, NULL, NULL);
46
47 /* Build the program */
48   clBuildProgram(program,1,&(settings->device_id), NULL, NULL,
         NULL);
49
50 /* Create the kernel object */
51   cl_kernel kernel = clCreateKernel(program, "flexible_kernel",
         NULL);
52
53 /* individual settings - CPU/GPU threads usage */
54   int total_thrd[2] = {(settings->global1),(settings->global2)};
55   int group_thrd[2] = {(settings->local1), (settings->local2)};
56
57 /* connect the output m-elem ports to the corresponding results */
58 <foreach p in I_multi_out>
59   <p.name>->data = (unsigned char*) result_<p.name>;
60 <endforeach>
```

As described in Section II, there are various platforms with different GPU architectures. In order to access the appropriate

OpenCL function, a common API, presented in our previous work [13], is defined to abstract the different hardware characteristics through a set of functions. For example, the *apiCreateBuffer* used in line 3 to allocate buffer memory, based on the hardware and software capabilities (i.e., the device memory features and the existing OpenCL version on the platform), allocates memory on the appropriate location using the appropriate OpenCL function.

Listing 3: Behavior function

```
1  /* Set the arguments of the kernel */
2  <counter=0>
3  <for each p in I_reg_in>
4    apiSetKernelArg(kernel,<counter>, sizeof(<p.type>), (void*)&
         <p.name>);
5    <counter+= 1>
6  <endforeach>
7  <for each p in I_reg_out>
8    apiSetKernelArg(kernel,<counter>, sizeof(<p.type>), (void*)&
         result_<p.name>);
9    <counter+= 1>
10 <endforeach>
11 <foreach p in I_multi_in>
12   apiSetKernelArg(kernel,<counter>, <p.width*p.height*p.size>, (
         void*)&<p.name>);
13   <counter+= 1>
14 <endforeach>
15 <for each p in I_multi_out>
16   apiSetKernelArg(kernel,<counter>, <p.width*p.height*p.size>, (
         void*)&result_<p.name>);
17   <counter+= 1>
18 <endforeach>
19
20 /* Execute the OpenCL kernel */
21   clEnqueueNDRangeKernel(settings->cmd_queue, kernel, 2, NULL,
         total_thrd, group_thrd, 0, NULL, NULL);
22
23 /* Wait for command queue and GPU to finish their activities */
24   clFlush(settings->cmd_queue);
25   clFinish(settings->cmd_queue);
26
27 /* copy regular kernel output to correpsonding output port */
28 <foreach p in I_reg_out>
29   apiEnqueueReadBuffer(command_queue, result_<p.name>, CL_TRUE, 0,
         sizeof(<p.type>), &<p.name>, 0, NULL, NULL);
30 <endforeach>
```

The *behavior function*, described in Listing 3, is generated as follows. Using the memory buffers of the input and output data ports, the arguments of the kernel function are automatically provided (lines 4, 8, 12 and 16). The kernel execution is described in line 21, while in the last part (i.e., line 24 and 25), specific OpenCL functions are used to wait for the execution completion of the functionality. The last line of the function (i.e., 29) describes how the data from result buffers (with one value) are copied to the corresponding output regular ports.

The *destructor* releases the component resources. Listing 4 describes the destructor code, where the environment information (i.e., kernel object and program) are released (lines 2 and 3). The destructor also releases the memory buffers that were initially allocated by the constructor (lines 5 and 9).

Listing 4: Destructor code

```
1  /* Clean up */
2    clReleaseKernel(kernel);
3    clReleaseProgram(program);
4  <foreach p in I_out>
5    apiReleaseBuffer(result_<p.Name>);
6  <endforeach>
```

B. System-level realization

To facilitate the component communication at the system level, we use the concept of adapters proposed in our previous work [5] [6]. In the following paragraphs, we present in details how adapters are utilized in the context of flexible components.

1) Adapter identification. Components executed by different processing units (i.e., CPU and GPU) need, for particular platforms, data transferred between memory systems. For example, data send by a regular component to a GPU-allocated flexible component needs to be copied to the GPU memory system in order to be accessed by the flexible component. Instead of being encapsulated in the flexible components, the transfer activities are lifted from the component level and automatically provided through transparent adapters, based on the platform specifications.

There are two types of adapters corresponding to the transfer activity directions, i.e., CPU-to-GPU and GPU-to-CPU adapters, each one accomplishing an one-to-one port communication between two components. In our approach, adapters are implemented as regular components by following the component model specification, and are automatically generated by fulfilling three generation steps: *I)* introduce the adapters, *II)* remove unnecessary adapters, and *III)* merge the adapters where possible. The steps are presented as follows.

Step I. Two rules define the introduction of adapters:

a) Two components, of which at least one is a flexible component, communicate through data ports of the multi-element type.

b) One of the components, which is a flexible component, is allocated to the GPU and the other component (either regular or flexible) is allocated to the CPU.

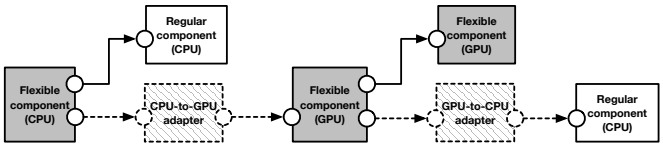


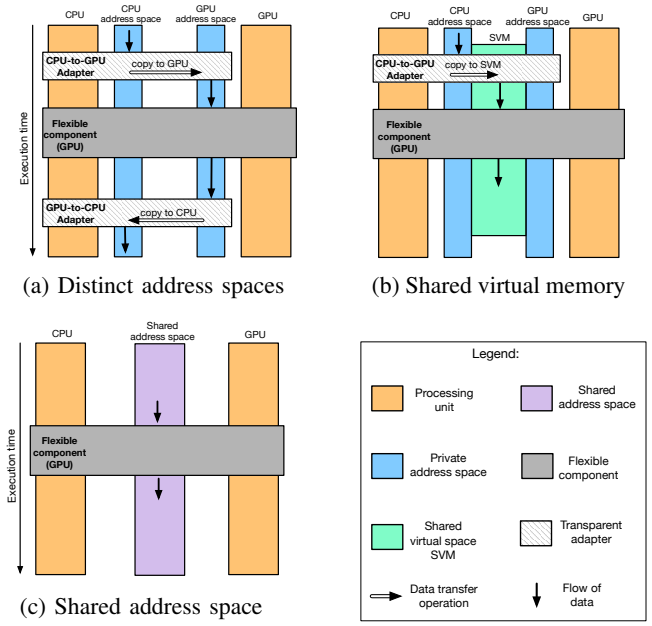
Fig. 2: Basic generation of adapters

We exemplify the generation of adapters in Fig. 2, where a CPU-to-GPU adapter facilitates the communication between CPU- and GPU-allocated flexible components. Similarly, a GPU-to-CPU adapter facilitates communication between GPU- and CPU-allocated components. No adapters are required when connected components are allocated on the same processing units.

Step II. The hardware characteristics impose additional restrictions on the generation of adapters, as follows:

- If the platform has a full-shared memory system, there will be no generation of adapters (due to the direct memory access feature).
- For platforms with shared virtual memory, only CPU-to-GPU adapters are generated.

Fig. 3 illustrates the generation of adapters for platforms with different characteristics. In Fig. 3(a) where the platform has distinct address spaces, generated adapters transfer data to appropriate location. For platforms with shared virtual memory support (Fig. 3(b)) there is need only for CPU-to-GPU adapters to specifically transfer data on the SVM space.



(a) Distinct address spaces (b) Shared virtual memory

(c) Shared address space

Fig. 3: Adapters in different hardware architectures

For this particular platform, there is no need for GPU-to-CPU adapters because all components can directly access the SVM space. The last case where platforms have full shared memory (Fig. 3(c)), there is no generation of adapters; all CPU- and/or GPU-allocate components can directly access the memory.

Step III. For optimization purposes, we group several (one-to-one into one-to-many) adapters, using the following rules:

- when a regular or CPU-allocated flexible component communicates with several GPU-allocated flexible components through one output port, the communications are done through a single CPU-to-GPU adapter.
- when a GPU-allocated flexible component communicates with several regular or/and CPU-allocated flexible components through one output port, the communications are done through a single GPU-to-CPU adapter.

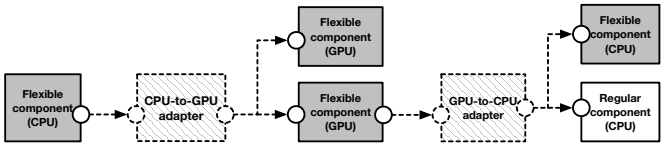


Fig. 4: Optimization of generation of adapters

The reason to have one adapter is that, instead of copying several versions of the same data on the main/GPU memory system, we transfer the data once and provide it to several components. Fig. 4 exemplifies the optimization step, where one CPU-to-GPU adapter transfers data on the GPU memory system and provides it to two GPU-allocated flexible components. Similarly, a GPU-to-CPU adapter copies once data on main memory system.

2) Adapter generation. This section describes the generation of the adapters' constituent parts, i.e., constructor, behavior function and destructor, as follows.

The constructor. The adapter has one input data port *p_in* and one output data port *p_out*, both of multi-element type. The adapter's constructor allocates memory (line 2) corresponding to the size of input data, on the appropriate location, i.e., the GPU (for CPU-to-GPU adapters) or main memory (for GPU-to-CPU adapters). The output port is linked to the location that holds the copied input data (line 5).

Listing 5: Constructor code of adapters

```
1  /* create memory buffers for the (one) output port */
2  void *result_adp = apiCreateBuffer(settings->contex,
       CL_MEM_WRITE_ONLY, <p_in.width*p_in.height*p_in.size>,NULL,
       NULL);
3
4  /* connect the output port to the created buffers */
5  <p_out.name>->data = (unsigned char*) result_adp;
```

Behavior function. The generated code of this part handles the transfer of data to or from GPU, corresponding to the hardware allocation of the connected components. The *clEnqueueWriteBuffer* is synchronous (i.e., returns the control after it finishes) due to the usage of *CL_TRUE* flag.

Listing 6: The adapter behavior function

```
clEnqueueWriteBuffer(settings->cmd_queue, result_adp, CL_TRUE, 0,
      <p_in.width*p_in.height*p_in.size>, <p_in.name>->data, 0, NULL,
      NULL);
```

The destructor. Opposite to the adapter's constructor that allocates one memory space for the input data, the adapter's destructor releases this memory.

Listing 7: Destructor code of adapters

```
/* Clean up */
apiReleaseBuffer(result_adp);
```

V. Feasibility Evaluation

This section presents a feasibility evaluation of our solution. The first part describes the underwater robot vision system used in the case study, while the rest of the section examines our solution implementation and its output for different cases.

A. Case study

For the evaluation, we use the vision system of an underwater robot as a case study. The robot autonomously navigates under water with the purpose to fulfill various missions that involves navigating based on vision, e.g., identifying red buoys [14]. The hardware platform contains an electronic board with a GPU, that is connected to various sensors (e.g., two cameras) and actuators (e.g., thrusters). The continuous flow of data produced by the cameras, is processed by the robot's vision system using the GPU.

Fig. 5 presents a simplified Rubus version of the component-based vision system. Two *Camera* components are connected to the physical camera sensors. The received data is converted into readable frames and forwarded to the *MergeAndEnhance* component that merges and reduces the noise of the two received frames. The resulting frame is converted to grayscale

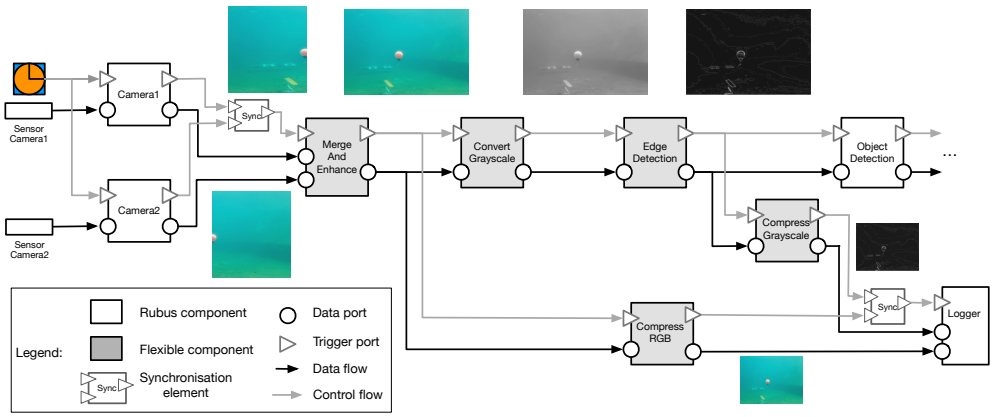


Fig. 5: The vision system of the underwater robot

by *ConvertGrayscale*, and passed to *EdgeDetection* component that outputs a frame where the object edges are presented with white color on a black background. The *ObjectDetection* component, based on the received black-and-white frame, detects the existing objects. To record the robot's underwater journey e.g., for debugging purposes, the *Logger* component registers the compressed underwater frames, both the original (merged) and black-and-white frames. Due to the nature of computations (i.e., image processing), we use *MergeAndEnhance*, *ConvertGrayscale*, *EdgeDetection*, *CompressRGB* and *CompressGrayscale* as flexible components. The frames are of *m-elem* type, where the maximum size (i.e., RGB and grayscale) varies from component to component, depending on the functionality.

Listing 8 illustrates the functionality code of the *ConvertGrayscale* flexible component that receive a 2D color image through the input multi-element port named *ImgIn*. To access the width and height dimensions of the input data, the developer uses the macros (see Section IV-A) as illustrated in e.g., lines 6 and 10. 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 14, 15 and 16). Finally, each output pixel is initialized with its grayscale value (line 21).

Listing 8: *ConvertGrayscale* kernel code

```
1  /* compute absolute image position (x, y) */
2    int row = get_global_id(0);
3    int col = get_global_id(1);
4
5  /* relieve any thread that is outside of the image */
6    if(row >= ImgIn_width || col >= ImgIn_height)
7      return;
8
9  /* compute 1-dimensional pixel index */
10   int index = row + ImgIn_width * col;
11
12 /* load RGB values of pixel (converted to float) */
13   float3 pixel;
14   pixel.x = ImgIn_data[ImgIn_size * index];
15   pixel.y = ImgIn_data[ImgIn_size * index + 1];
16   pixel.z = ImgIn_data[ImgIn_size * index + 2];
17
18 /* compute luminance and store to output array */
19   float lum = 0.2126f*pixel.x + 0.7153f*pixel.y + 0.0721f*pixel.z;
20
21   ImgOut_data[index] = (unsigned char)lum;
```

B. Experimental setup

To evaluate our approach, we use four allocation scenarios for the vision system, described in Table I. In scenario 1 all flexible components are allocated to the GPU; in scenario 2, all flexible components are allocated to the CPU. In scenario 3 and 4, we alternate in allocating the flexible components to CPU and GPU.

TABLE I: Different allocation scenarios for the vision system

Flexible Component	Hardware allocation			
	Scenario 1	Scenario 2	Scenario 3	Scenario 4
MergeAndEnhance	GPU	CPU	CPU	GPU
ConvertGrayscale	GPU	CPU	GPU	CPU
EdgeDetection	GPU	CPU	CPU	GPU
CompressRGB	GPU	CPU	GPU	CPU
CompressGrayscale	GPU	CPU	CPU	GPU

Moreover, for each scenario, we 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[6], and *c)* an embedded platform with an AMD Carizzo SoC with full shared memory architecture[6].

Depending on the flexible component allocation and the hardware characteristics, the resulting vision system is populated with adapters. Fig. 6 illustrates the vision system supplied with adapters to address scenario 1 for the dGPU-based platform. In this case, five adapters are generated illustrated as dash-line components (i.e., two CPU-to-GPU adapters and three GPU-to-CPU adapters).

C. Results

We compared three produced frames (i.e., the input to *ObjectDetection* and *Logger*) from all twelve combinations of scenarios and platforms; all combinations generated identical output frames.

In addition, we present the number of generated adapters for all cases (i.e., four scenarios and three platforms), in Table II.

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

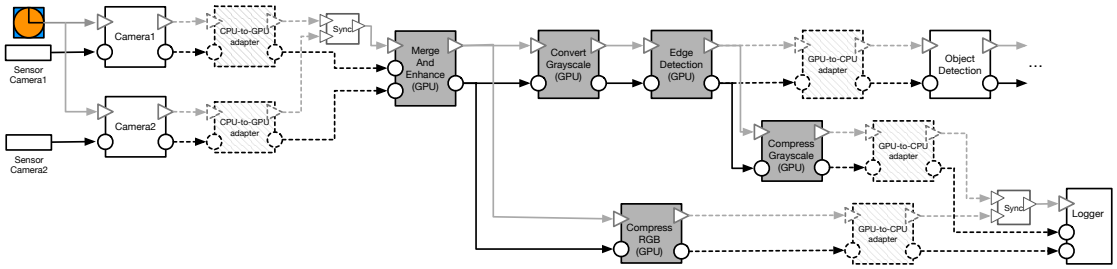


Fig. 6: Vision system with adapters for dGPU-based platforms

For scenario 1 (i.e., all flexible components allocated on GPU), for platforms with dGPU architecture, there are generated two CPU-to-GPU adapters and three GPU-to-CPU adapters (see Fig. 6). When all flexible components are allocated to CPU, there is no need for adapters. For shared virtual memory architectures, there are generated only CPU-to-GPU adapters; there is no need for GPU-to-CPU adapters because all components (regular and flexible) have direct access to the same shared virtual memory system.

TABLE II: Experimental results

Scenario	Platform type	Number of adapters		Code size	
		CPU-to-GPU	GPU-to-CPU	Generated	Written
1	dGPU	2	3	7819	4101
	iGPU1	2	0	6871	
	iGPU2	0	0	6239	
2	dGPU	0	0	6239	4101
	iGPU1	0	0	6239	
	iGPU2	0	0	6239	
3	dGPU	1	1	6878	4101
	iGPU1	1	0	6554	
	iGPU2	0	0	6239	
4	dGPU	1	3	7503	4101
	iGPU1	1	0	6554	
	iGPU2	0	0	6239	

dGPU - CPU and GPU distinct memory platform
iGPU1 - Shared Virtual Memory platform
iGPU2 - Shared Memory platform

The table also describes the size of the generated code in comparison to the manually written code. The generated code, including all flexible components and adapters, varies from 6239 characters for shared memory platforms to 7819 characters for platforms with dGPUs. For all scenarios and platforms, the flexible component functionalities are the same, i.e., a total of 4101 characters manually written.

VI. Related Work

The first part describes programming models for different platforms. The second part presents existing component-based solutions and how they address system flexibility.

A. Programming models for heterogeneous platforms

The market contains programming models that specifically target GPUs. We mention CUDA[7] that targets only NVIDIA GPUs, CTM[8] for ATI AMD GPUs, and Brook [15]. OpenCL bridges the gap between CPU and GPU, being employed even further by other processing unit types such as FPGA. However, with OpenCL, the communication between processing units falls under the developers responsibility. EXOCHI [16] and Merge [17] hide the communication responsibility from the developer and provide uniform frameworks that target different processors. Using them, the developer needs to write different versions of the same functionality, for different processing units. We also mention frameworks that automatic translate between CPU and CUDA code. MCUDA [18] translates CUDA code into multi-thread CPU code, and the framework proposed by Lee et al. [19] translates OpenMP code to CUDA code. An improved framework, i.e., MapCG [20], contains a high level API (C-like language) that hides programming side-burdens (e.g., communication). Similarly, we provide transparent communication mechanism (i.e., adapters) that ease the development responsibility.

Although the enumerated solutions are not intended to be used in a CBD context, they may be employed in out work for hiding the communication overhead and eliminating adapters. Their disadvantage is that they may introduce additional overhead and resource utilization which represents an important factor for the embedded systems targeted by our work.

It is worth to mention the Aspect-Oriented Programming (AOP) paradigm where various concerns called *aspects*, are inserted in the source code. For example, Wang et al. [21] propose a programming system to assist the development of GPU functionality. More exactly, a special compiler inserts GPU aspect code fragments (e.g., memory transfer activities) in the C++ source code, resulting in GPU applications. Similarly with AOP, we generate parts with GPU-specific information for flexible components and adapters. However, the AOP paradigm does not provide the encapsulation concept that is required in our solution.

B. Heterogeneous component-based systems

Flexibility for embedded systems is addressed in different ways and for different reasons. For example, Lednicki et al. [22] tackled the way that some component models develop systems (i.e., hard-coding inside the software component the hardware platform characteristics) by introducing an additional layer (i.e., mapping layer). The mapping layer connects the

[7]http://docs.nvidia.com/cuda/

[8]http://roland.pri.ee/doktor/papers/gpgpu/ATI_CTM_Guide.pdf

software and hardware layers, allowing them to be developed independent of each other, improving the component reusability for different (hardware and software) contexts. Although aiming at a different platform type (i.e., with GPU) that brings particular challenges, our solution also increases the reusability of flexible components for different hardware platforms.

Regarding platforms with various processing units (e.g., GPUs, FPGAs), elastic computing framework [23] allows a transparent application design by selecting among different implementations (referred as elastic functions), the most efficient one. The elastic functions should be developed separately and included in the framework. Although is not a component model per se, the elastic computing framework has similar principles (e.g., interfaces, usage of already developed functions). Using the same approach, the PEPPHER model contains different C/C++ based components (i.e., sequential and parallel implementation) for the same functionality, that can be executed on systems with or without GPUs. The data that is passed between PEPPHER components is encapsulated in containers; there are three types of containers (i.e., scalar, array and 2D array) to support different type of data. Opposite to these approaches, our work's advantage is that it has less overhead (e.g., memory footprint, development time) by using a single component version that can be executed either on CPU or GPU hardware. This component functionality is written in a generic way, so the performance achieved by our solution may be inferior to the one provided by the elastic and PEPPHER frameworks. In addition, as the PEPPHER model has dimension-specialized data-containers, we use the same adapter to take care of data transfer between components, regardless of their dimension size (i.e., 1, 2 or 3 dimensions).

VII. Conclusions

The existing embedded boards with GPU capabilities provide feasible solutions for the stringent requirements of modern embedded systems. CBD methodology, successfully integrated in industry for development of traditional embedded systems, provide no specific support when it comes to platforms with GPUs. In this context, we introduce flexible components to address parts of the CBD lack of GPU support.

Through our solution, the component developer is released from side activities which are automatically handled. Employing flexible increases the design-solution space which may improve the system overall feasibility. Moreover, we facilitate the component communication by utilizing specialized artifacts (i.e., adapters) that automatically transfer data between CPU- and GPU-allocated flexible components.

Acknowledgments

The Swedish Foundation for Strategic Research (SSF) supports our work through the RALF3 project (IIS11-0060).

References

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

[2] "AUTOSAR - Technical Overview," http://www.autosar.org, accessed: 2017-03-24.

[3] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "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.

[4] 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.

[5] 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*, 2015.

[6] 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, pp. 59–68.

[7] M. Chaudron and I. Crnkovic, "Component-based software engineering," in *Software Engineering: Principles and Practice*, H. van Vliet, Ed. Wiley, 2008.

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, 2008.

[10] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *Signal Processing and Communications IEEE International Conference*. IEEE, 2007, pp. 65–68.

[11] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468 – 4477, 2009.

[12] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of computational chemistry*, 2007.

[13] G. Campeanu, J. Carlson, and S. Sentilles, "Developing CPU-GPU embedded systems using platform-agnostic components," in *43rd Euromicro Conference on Software Engineering and Advanced Applications*, August 2017, to appear.

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

[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *ACM Transactions on Graphics (TOG)*. ACM, 2004.

[16] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *ACM SIGPLAN Notices*. ACM, 2007.

[17] 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*. ACM, 2008.

[18] J. A. Stratton, S. S. Stone, and W. H. Wen-mei, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 16–30.

[19] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[20] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: writing parallel program portable between CPU and GPU," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 217–226.

[21] M. Wang and M. Parashar, "Object-oriented stream programming using aspects," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[22] 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.

[23] J. R. Wernsing and G. Stitt, "Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '10. ACM, 2010, pp. 115–124.