Allocation Optimization for Component-based Embedded Systems with GPUs

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

Abstract—**Platforms equipped with GPU processors help mitigating the ever-increasing computational demands of modern embedded systems. Such systems can be specifically developed by using component-based development thanks to the concept of flexible components. Through this concept, a component can be transparently executed either on a CPU or a GPU. However, this flexibility complicates the allocation process, i.e., it is more difficult to know where to allocate the components to best utilize the limited resources of the platform. In this work, we address this problem by providing an optimization model for component-based embedded systems executing on both CPU and GPU. The model addresses important optimization goals, characteristic to the embedded system domain, such as memory usage, energy usage and execution time. A novelty of this work is the formal description of the optimization model, which supports the usage of mixed integer nonlinear programming to compute optimal allocation schemes. To examine the feasibility of the proposed method, we apply the optimization model on a vision system constructed using the industrial Rubus component model.**

Index Terms—**Optimization, component allocation, flexible component, embedded systems, CBD, component-based development, GPU.**

I. Introduction

The new boards with general-purpose Graphics Processing Units (GPUs) are feasible solutions employed in embedded systems to tackle the demanding requirements of modern applications. Equipped with a parallel execution model, the GPU is a processing unit that excels in handling the huge amount of information, which, for some embedded systems, often originates from e.g., the interaction with the environment.

Another trend in embedded systems is the usage of component-based development (CBD). This software engineering paradigm promotes the construction of applications through the composition of already existent software blocks called *software components*. Nowadays, CBD is successfully integrated in industry, and also in the embedded systems domain, through state-of-the-practice component models such as AUTOSAR [1], IEC 611-31 [2] and Rubus [3].

Our previous work facilitates the development of embedded systems with GPUs via two complementary concepts, i.e., the *flexible component* and *adapter* [4]. A flexible component has the functionality that can be indifferently executed by either CPU or GPU. This concept exists at design time, when the system designer decides where to place the flexible components for execution (i.e., either on CPU or GPU) in order to achieve the required system performance. Once the allocation has been decided, adapters are automatically generated to facilitate communication between components allocated on different processing units.

After designing the system, an important step is the allocation process, i.e., how to allocate functionality over the hardware in order to utilize, in the best way, the limited resources of the embedded platform. Deciding which functionality should be placed on a given processing unit is an NP-hard problem referred to as software deployment [5]. In the context of embedded systems with GPUs, the complexity is further increased due to: *i)* the heterogeneity of the hardware (i.e., platforms with CPU and GPU) and, *ii)* the flexible components (i.e., components with variable allocation) and adapters (i.e., automatically generated artifacts that connect components allocated on different processing units).

This paper presents a method that automatically provides suitable allocation schemes for component-based systems. The schemes are optimized w.r.t. particular system goals, and describe the flexible component allocations on the CPU or GPU. The proposed method receives as input the system model that describes the interconnected components and their characteristics, and the platform properties. The considered optimization goals are related to the focused domain of this work (i.e., embedded systems) and address the memory usage, energy usage and execution time of the system.

An optimization problem may be solved using a heuristic method such as genetic algorithms, or exact methods such as the mixed integer nonlinear programming (MINLP). While heuristics offer approximate solutions in shorter calculation times, exact methods provide optimal solutions for formally-defined optimization models, but with a longer computation time for complex problems [6]. A part of the novelty of this work is in the formal definition of the optimization model, allowing in this way, to employ MINLP for solving it. To examine the feasibility of the approach, we applied the optimization model on a vision system constructed using the state-of-the-practice Rubus component model, and solved it with a MINLP solver.

The reminder of the paper is divided as follows. The background of GPUs and CBD in the context of embedded systems is presented in Section II followed by the solution overview in Section III. The formalized optimization model is covered by Section IV. We introduce a case study in Section V, which is used to examine the feasibility of the optimization model. The related work is described in Section VI, followed by conclusions in Section VII.

II. GPUs and CBD in embedded systems

This section presents, in two parts, the background of this work, i.e., GPUs (in the first part) and CBD (in the second part). As our focus is in the embedded systems domain, the background notions are introduced in the targeted domain.

A. GPUs in embedded systems

GPUs were initially developed for graphics-based applications. Over time, GPUs have been equipped with more resources and, being easier to program, has led to them being employed in general-purpose applications.

The usage of GPUs has been adopted in industry as a way to tackle the stringent requirements (e.g., limited energy usage) of modern applications. Nowadays, embedded platforms with GPUs are feasible solutions in e.g., providing the required performance when processing the data produced by the system's sensors. For example, DRIVE PX Xavier[1] is a platform developed by NVIDIA to be used in the automotive domain, for autonomous transportation applications.

Today, there are many embedded platforms with GPUs, with different characteristics. Various vendors such as NVIDIA, Intel, AMD, Xilinx and IBM, provide their solutions regarding the resources and physical architecture of the platforms with GPUs. One of these architectures is represented by platforms with the GPU and CPU integrated on the same chip. The reduced physical size, energy usage and cost make this type of architecture to be the most used in industry.

A characteristic of this type of architecture is the distinct CPU and GPU memory addresses. Although both processing units are integrated on the same physical chip, the memory is divided in two distinct memory address spaces, one for each processing unit. As an effect, the data needs to be transferred back and forth between the memory addresses in order to be accessed by the CPU or GPU.

B. Component-based development

Component-based development is a software engineering methodology that promotes the development of applications via composition of already existing software blocks referred as *software components*. CBD has been successfully adopted by industry through several state-of-the-practice component models that specifically target the development of embedded systems, such as AUTOSAR [1], IEC 611-31 [2] and Rubus [3]. A common characteristic of component models used in development of embedded and real-time systems, is that they typically follow a *pipe-and-filter* interaction style. This particular interaction style allows an easy mapping of the control requirements in the domain.

Rubus is a state-of-the-practice component model used by e.g., Volvo Construction Equipment to develop embedded software. The Rubus component, called *software circuit*, follows a Read-Execute-Write semantics, i.e., when it receives the control, it reads the data from the input data ports, executes the behavior, writes the results to the output data ports and finally hands off the control to the next connected component. Another characteristic of the component model is the distinction between data and control flow. Fig. 1 presents two connected Rubus components and their control and data flow description. The element that triggers *C1* is a periodic trigger source (i.e., a *clock*), characterized by a particular activation period (e.g., 2 ms).

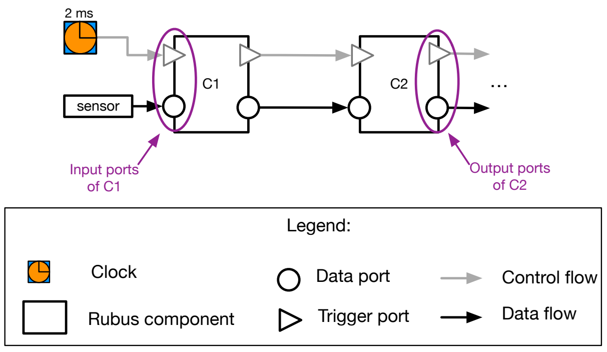

Fig. 1: Connected Rubus components

Up until recently, there has been limited work in CBD w.r.t. dedicated GPU support. We introduced the concepts of *flexible component* and *adapter* to facilitate the development of embedded systems with GPUs [4]. A flexible component has a functionality that can be executed by either CPU or GPU. The flexibility, however, only exists at design-time. Based on the allocation, decided at design-time, the flexible component is realized as a regular component that encloses all the information required to be executed on the allocated hardware.

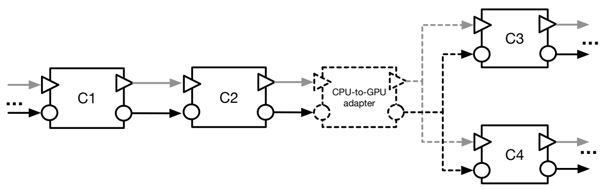

Fig. 2: Adapter connecting Rubus components allocated on different processing units

The adapter is a concept that complements the flexible component notion, and decreases the development complexity and error-proneness. Due to the platform characteristics, data needs to be transfered between CPU and GPU memory spaces, in order to be accesses by the components. The data transfer is automatically achieved by adapters, which are seen as artifacts that connect components allocated on different processing units. While the flexible component concept exists at design-time, the adapter is introduced in a later stage, i.e., the system realization stage. Based on the allocation of the flexible components, adapters are automatically generated where needed, realized as components that transfer data between the CPU and GPU memory addresses. Fig. 2 illustrates a CPU-to-GPU adapter that facilitates the communication between the CPU-allocated *C2*, and GPU-allocated *C3* and *C4*. The adapter

[1]https://blogs.nvidia.com/blog/2016/09/28/xavier/

transfers data from the CPU to the GPU memory space, in order to be accessed by *C3* and *C4*. There is no need for adapter between *C1* and *C2* as both are allocated on the CPU, accessing the same memory address.

III. Overview of the approach

In the context of embedded systems with flexible components, a challenge appears when determining the allocation of the flexible components with respect to certain optimization goals. Moreover, the challenge is also increased by the fact that, at the realization of the allocation solution, adapters are generated for data transfer activities, and these adapters influence the suitability of the allocation due to e.g., their memory usage and energy consumption.

In this paper, we propose a method that automatically computes optimized allocation schemes for a component-based embedded system with GPU. As input, the method receives the system model (capturing relevant properties of both software and hardware), and the optimization criteria.

In the rest of the section, we give high level descriptions of the system model, the optimization criteria and the allocation result. Details about the how these concepts are represented in the optimization problem, are presented in Section IV.

A. System model

The system model includes both software and hardware aspects. The software is represented by connected components that follow the pipe-and-filter interaction style. We distinguish two types of components used: i) regular components with CPU functionality, and ii) flexible components with functionality that can be executed either on CPU or GPU.

We consider that each component is characterized by the following three properties when allocated on the CPU:

- The *internal memory usage* represents the CPU memory requirement of a component to properly execute its functionality. The requirement refers to the internal memory used by the behavior, such as the variables defined in the functionality, excluding memory for the data ports.
- The *energy usage* describes the amount of energy spent by a component to execute its functionality.
- The *execution time* specifies the time required by a component to execute its functionality in isolation.

When a component is allocated on the GPU, it is characterized by four properties, as follows:

- The *internal GPU memory usage* represents the component requirement of GPU memory.
- The *internal CPU memory usage* represents the CPU memory requirement. Besides the GPU memory requirement, a GPU-allocated component may also contain variables that reside in the CPU memory space.
- The *energy usage* is the amount of energy used in the component execution.
- The *execution time* describes the component execution time on GPU.

Each component is characterized by the two previously described property sets, i.e., one for the CPU and the other for the GPU. Regular components, being only CPU executable (i.e., allocated on CPU), have the GPU-related properties set to zero.

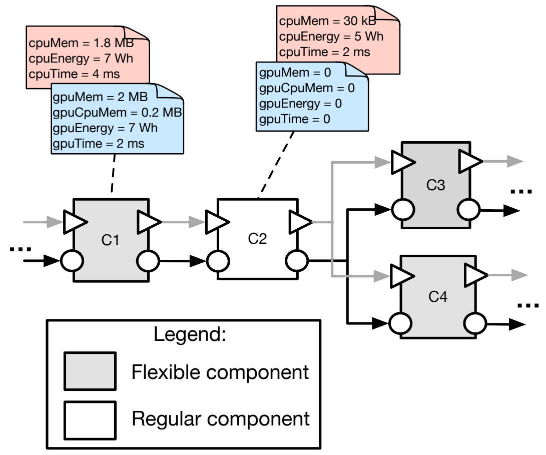

Fig. 3: Connected (flexible and regular) Rubus components

Fig. 3 presents a section of a Rubus application composed of a regular (i.e, *C2*) and three flexible components (i.e., *C1*, *C3* and *C4*). As illustrated by the figure, a flexible component has two sets of properties with non-zero values. The component *C2* is characterized by the properties related to the CPU such as it executes on CPU for 2 milliseconds and consumes 5 Watthours; the GPU-related properties are set to zero.

The hardware platform contains a CPU coupled with a GPU, where each processing unit has its private memory address space. The properties that characterize the hardware platforms are: *i)* the available GPU memory, and *ii)* the available CPU memory.

B. Constraints and optimization criteria

An allocation scheme is decided based on the properties of the application, the characteristics of the platform and the system requirements. A given system may have several feasible allocations. However, not all allocations are equivalent, i.e., there are trade-offs when selecting an allocation over the other. For example, one feasible allocation may optimize the system memory utilization to the detriment of the performance. Therefore, it is important to decide, from all feasible allocations, which one is suitable for a given system.

A suitable allocation is determined by the constraints and optimization criteria. The constraints need to be satisfied in order for the allocation to be feasible. For example, a constrain forces to place on GPU a number of components that together do not require more memory than is available. The optimization criteria are used to decide which allocation solution is better. In this work, we place ourself in the context of embedded systems and address related criteria to this domain. We introduce optimization criteria such as memory usage, energy usage and performance. For example, because many embedded systems have stringent requirements regarding the power consumption, a criterion for our allocation method is to minimize the energy usage of the system.

An important factor to be considered by the constrains and optimization criteria is the adapters (see Section II-B). These artifacts are generated to facilitate component communication between processing units, where needed. For instance, using the example presented in Fig. 3, if *C1* is decided to be allocated on the GPU, then its communication with the regular (CPU-allocated) *C2* component is facilitated via an adapter. The constraints and optimization criteria must not only consider the properties of the system components, but also the costs (e.g., energy usage) introduced by the generated adapters.

C. Allocation result

The allocation result is the product of the optimization process, and contains a feasible solution of the mapping of software components onto the processing units. The result satisfies the requirements introduced as input, i.e., the software characteristics and the hardware constrains. Focusing, in this work, on systems composed of flexible components, the allocation result presents which flexible components are allocated to the CPU and which are allocate to the GPU, in order to satisfy the optimization criteria. The allocation result is represented by a mapping between the (flexible) components and the processing units (i.e., CPU/GPU). For example, using the application described in Figure 3, one of the possible allocation results that minimizes the CPU memory usage is described as follows:

$$\begin{array}{ll} C1 \rightarrow GPU & C3 \rightarrow GPU \\ C2 \rightarrow CPU & C4 \rightarrow GPU \end{array}$$

IV. Optimization model

The proposed optimization model is formally defined in the following paragraphs. The model contains the formal definitions of the input and output, and the constraints and optimization criteria included in this work.

A. Input

The model of the system structure is described in the following paragraphs.

Let $C = \{c_1, \dots, c_n\}$ be a set of n components, divided into two disjoint subsets, $C = R \cup F$, representing regular and flexible components, respectively. Each component is characterized by one or several unique input data ports, and output data ports. Let $I = \{ip_1, \dots, ip_m\}$ be a set containing the input data ports of all the components of the system, and $O = \{op_1, \dots, op_k\}$ a set containing the output data ports of all the components of the system.

Moreover, we define the set $S = \{\langle f_1, C_1\rangle, \dots, \langle f_q, C_q\rangle\}$ describing the triggering of components, where $C_i \subseteq C$ is a subset containing the components that are triggered by the same unique trigger source (i.e., clock) that has the frequency of f_i. Each component from C must be in exactly one trigger set C_i, i.e., $C_1, \dots, C_q$ are disjoint, with $C_1 \cup \ldots \cup C_q = C$.

Based on the defined sets, we introduce two functions to describe the system structure (i.e., *comp* and *connect*), and eight functions to represent properties of individual components and ports. The definitions of the functions are described by Table I.

TABLE I: Ten functions describing the system model

Function	**Description**
$comp : I \cup O \rightarrow C$	$comp(p)$ = the component that has p as an input or output data port.
$connect : O \rightarrow 2^I$	$connect(op)$ = the set of input ports connected to *op*.
$portSize : O \rightarrow \mathbb{N}$	$portSize(p)$ = the size of data sent through the output port *op*.
$cpuMem : C \rightarrow \mathbb{N}$	$cpuMem(c)$ = the internal CPU memory usage of *c* when allocated to a CPU.
$gpuMem : C \rightarrow \mathbb{N}$	$gpuMem(c)$ = the internal GPU memory usage of *c*.
$gpuCpuMem : C \rightarrow \mathbb{N}$	$gpuCpuMem(c)$ = the internal CPU mem usage of *c* when allocated to a GPU.
$cpuTime : C \rightarrow \mathbb{N}$	$cpuTime(c)$ = the execution time of *c* on CPU.
$gpuTime : C \rightarrow \mathbb{N}$	$gpuTime(c)$ = the execution time of *c* on GPU.
$cpuEnergy : C \rightarrow \mathbb{N}$	$cpuEnergy(c)$ = the energy usage of *c* on CPU.
$cpuEnergy : C \rightarrow \mathbb{N}$	$cpuEnergy(c)$ = the energy usage of *c* on GPU.

Regarding the hardware platform, we introduce two variables to describe the platform characteristics:

$$AvailCpuMem = \text{available CPU memory}$$
$$AvailGpuMem = \text{available GPU memory}$$

Moreover, we define two constants as follows:

- k_{eng} to describe the energy usage of transferring one unit of data between CPU and GPU memory addresses, and
- k_{tm} to describe the time used to transfer one unit of data between distinct memory addresses.

We assume that the same energy/time is used when transferring one unit of data from the CPU to GPU memory space, and vice-versa.

B. Output

The result is a scheme that contains the allocation of all flexible components. Let $A = \{a_{c_1}, \dots, a_{c_n}\}$ be a set of boolean variables, where each element a_{c_i} represents the mapping of the corresponding component c_i to a processing units.

$$a_{c_i} = \begin{cases} 0, \text{ when } c_i \text{ is allocated on CPU} \\ 1, \text{ when } c_i \text{ is allocated on GPU} \end{cases}$$

C. System properties

We define four system properties, as illustrated in Table II:

TABLE II: System properties

$$GpuMemory = \sum_{c_i \in C} a_{c_i} * gpuMem(c_i) + \sum_{op_i \in O} min\Big(1, a_{comp(op_i)} + \sum_{ip_j \in connect(op_i)} a_{comp(ip_j)}\Big) * portSize(op_i) \tag{1}$$

$$\begin{aligned} CpuMemory = &\sum_{c_i \in C} \Big((1 - a_{c_i}) * cpuMem(c_i) + a_{c_i} * gpuCpuMem(c_i)\Big) + \\ &\sum_{op_i \in O} min\Big(1, 1 - a_{comp(op_i)} + \sum_{ip_j \in connect(op_i)} \Big(1 - a_{comp(ip_j)}\Big)\Big) * portSize(op_i) \end{aligned} \tag{2}$$

$$\begin{aligned} Energy_{\langle f_l, C_l \rangle} = &\sum_{c_i \in C_l} \Big(a_{c_i} * gpuEnergy(c_i) + (1 - a_{c_i}) * cpuEnergy(c_i)\Big) + \\ &\sum_{op_i \in O \wedge comp(op_i) \in C_l} \Big(a_{comp(op_i)} * min\Big(1, \sum_{ip_j \in connect(op_i)} \Big(1 - a_{comp(ip_j)}\Big)\Big) + \\ &(1 - a_{comp(op_i)}) * min\Big(1, \sum_{ip_j \in connect(op_i)} a_{comp(ip_j)}\Big)\Big) * portSize(op_i) * k_{eng} \end{aligned} \tag{3}$$

$$Energy = \sum_{\langle f_l, C_l \rangle \in S} Energy_{\langle f_l, C_l \rangle} * f_l \tag{4}$$

$$\begin{aligned} Time_{\langle f_l, C_l \rangle} = &\sum_{c_i \in C_l} \Big(a_{c_i} * gpuTime(c_i) + (1 - a_{c_i}) * cpuTime(c_i)\Big) + \\ &\sum_{op_i \in O \wedge comp(op_i) \in C_l} \Big(a_{comp(op_i)} * min\Big(1, \sum_{ip_j \in connect(op_i)} \Big(1 - a_{comp(ip_j)}\Big)\Big) + \\ &\Big(\Big(1 - a_{comp(op_i)}\Big) * min\Big(1, \sum_{ip_j \in connect(op_i)} a_{comp(ip_j)}\Big)\Big) * portSize(op_i) * k_{tm} \end{aligned} \tag{5}$$

1) $GpuMemory$: the total GPU memory usage of the system. This property, presented by equation 1, describes the amount of GPU memory required by the system components. $GpuMemory$ is composed of two members, as follows. The first member sums the internal GPU memory usage of only the (flexible) components that are allocated on the GPU (i.e., components c_i that have the allocation a_{c_i} as 1).

The second member addresses the GPU memory usage of: *i)* the output ports of GPU-allocated components, and *ii)* possible adapters that transfer data from the CPU to GPU memory space. The output ports of components with GPU capability, are used to pass large data with multiple elements (e.g., 2D images), and have an important impact on the components' GPU memory requirement. Another aspect that affects the system's GPU memory usage is the fact that, when a CPU-allocated component communicates with at least one GPU-allocate component, the sent data is copied, by a late realized adapter, from the CPU to the GPU memory space (see Figure 2).

These aspects are captured by the second member of the equation, which uses a *min* function to verify both aspects, as follows. For each output port op_i of the O set:

- if it belongs to a GPU-allocated component (i.e., $a_{comp(op_i)}$ is 1), the *min* function produces the value 1, regardless of the connections of the output port.
- if it belongs to a CPU-allocated (flexible or regular) component (i.e., $a_{comp(op_i)}$ is 0), then the port's connections are verified in order to examine if an adapter would be generated. This situation is captured by the *sum* member inside the *min* function, where, if, at least, one connected port (i.e., ip_j) belongs to a GPU-allocated (flexible) component, than an adapter exists. In this case, the *min* produces the value 1.

To calculate the memory requirement of an output port or adapter, the value computed by the *min* function is multiplied with the port size provided via the *portSize()* function.

2) $CpuMemory$: the total CPU memory usage of the system. This property is calculated in a similar manner as the previous one, using two *sum* members, as follows. The first member adds, in the first part, the internal CPU memory requirement of the components that are allocated on the CPU (i.e., components c_i with $(1 - a_{c_i})$ as 1). Furthermore, in the second part of this member, it is added the CPU memory requirement of the components that are allocated on the GPU. The second member adds the CPU memory requirement of: *i)* the output ports of CPU-allocated (flexible) components, and *ii)* the adapters that transfer data from the GPU to CPU memory space.

3) $Energy$: the total energy usage of the system. This property is calculated by adding the energy usage of each set of components C_l triggered by the same source with the frequency f_l, as illustrated by equation 4. The energy usage of each of such component set C_l, described by equation 3, is obtained from two members, as follows.

The first member calculates the energy usage of the components in the set C_l by adding each of the components': *i)* GPU energy usage (i.e., provided by the $gpuEnergy()$ function), and *ii)* CPU energy usage (i.e., provided by the $cpuEnergy()$ function).

The second member calculates the energy spent in adapters transferring data between CPU and GPU memory spaces, as follows. The energy spent to transfer data from the GPU to the CPU memory space, illustrated in the first part of the member, takes each output port op_i of the components c_i that are allocated on CPU (i.e., $a_{comp(op_i)}$ is 1), and examines its connected (input) ports ip_j. The *min* function counts how many connected ports belong to CPU-allocated components. If there is at least one such connected port, an adapted will be generated for this transfer, and the *min* function returns the value 1. To calculate the energy spent on the transfer, the size of the output port (i.e., returned by the $portSize()$ function) is multiplied with the k_{eng} constant.

In a similar way, the second part of the member calculates the energy used on transferring data from the CPU to GPU memory space. The connections of each output port that belong to CPU-allocated components, are examined. If there is at least one connected port that belongs to a GPU-allocated component, then the energy used to transfer the data is given by the size of the output port multiplied with the k_{eng} constant.

4) $Time_{\langle f_l,C_l\rangle}$: the end-to-end execution time for a set of components C_l activated by the same trigger source. This property is expressed for each triggering of a group source. The execution time of a particular set of components C_l is formally defined in equation 5, through the sum of two *sum* members, described in the following paragraphs.

The first member calculates the execution times of the system components. The *sum* adds the GPU execution times of GPU-allocated (flexible) components and the CPU execution times of CPU-allocated (regular or flexible) components.

The second member deals with the time spent on transferring data between the distinct memory spaces, as follows. For the GPU-to-CPU transfers, the connections of the output ports of GPU-allocated (flexible) components are analyzed. If at least one of the connected ports belongs to a CPU-allocated (flexible or regular) component, an adapter will be generated, and its transferring time is calculated by multiplying the size of the output data port with the k_{tm} constant. In a similar way, the CPU-to-GPU transfer time is calculated in the last part of the second equation member.

D. Constraints

The optimization model considers four constraints related to the system properties, and one regarding the regular components, as follows:

1) GPU memory - the required GPU memory of components allocated on the GPU should not exceed the available GPU memory of the platform. Using the definition of the system's GPU memory usage (i.e., equation 1), the constrain is formally defined as:
$$GpuMemory \leq AvailGpuMem$$
2) CPU memory - similarly, the required memory of components allocated on the CPU should not exceed the CPU's available memory. Using the system's CPU memory usage (i.e., equation 2), the constrain is formalized as:
$$CpuMemory \leq AvailCpuMem$$
3) Maximum energy - the energy consumed by the system should be less or equal with a particular (maximum) limit. Using equation 4, the constrain is formalized as follows:
$$Energy \leq MaxEnergy$$
4) Maximum time - the execution time of a particular trigger group should be less or equal with a specific (maximum) execution time. The constraint is formalized using equation 5, as follows:
$$Time_{\langle f_l,C_l\rangle} \leq MaxTime_{\langle f_l,C_l\rangle}$$
5) All regular components must be allocated on the CPU:
$$a_c = 0 \text{ for all } c \in R$$

Because a system may be constructed from several sub-systems, the properties of each sub-system may be individually calculated using the equations defined in Table II. The constraint 3 may be used to delimit the energy usage of each separate sub-system, while constraint 4 delimits the execution times of groups enclosed by particular sub-systems. $MaxEnergy$ and $MaxTime_{\langle f_l,C_l\rangle}$ are user-defined constants. Similarly, constraint 1 and 2 my be used to delimit the memory limitations of each sub-system.

E. Optimization criteria

Our optimization is concerned with the following aspects:

- Minimize memory usage on the GPU.
- Minimize memory usage on the CPU.
- Minimize the system energy usage.
- Minimize the execution time of the system trigger groups.

In order to allow the developer to select which of the properties are more important, or to exclude one or several properties from the optimization process, *weight parameters* are introduced. We merge all of the concerns in a linear combination, and minimize the result, as follows:

$$\begin{aligned}
&minimize(F), \text{ where} \\
&F = w^g * GpuMemory + w^c * CpuMemory + \\
&\qquad w^e * Energy + \sum_{\langle f_l,C_l\rangle} w_l^t * Time_{\langle f_l,C_l\rangle}, \text{ and} \\
&w^g + w^c + w^e + \sum_{\langle f_l,C_l\rangle} w_l^t = 1
\end{aligned}$$

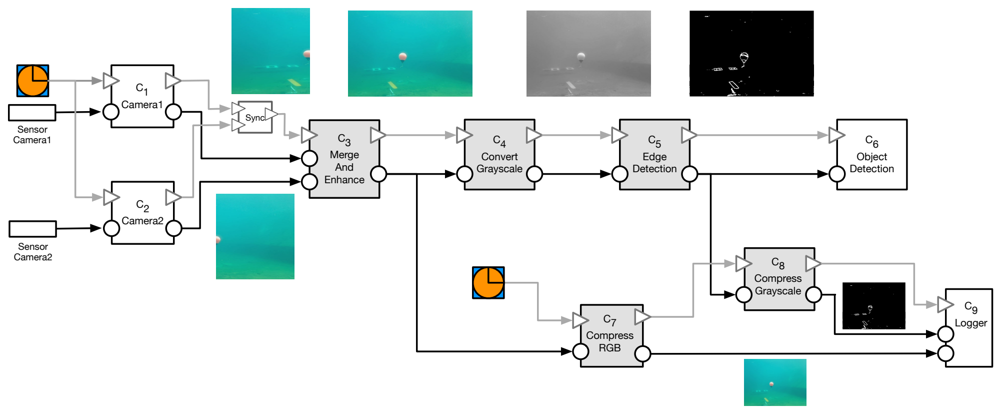

Fig. 4: The vision system of the underwater robot

F. Simplification of properties realization

The system properties described in Table II are formalized in a general manner, to cover systems with any number of components equipped with any number of ports, resulting in rather complicated equations due to the nested *sum* and *min* operations.

However, for any given system, the properties are calculated by unrolling the *sum* functions, leading to simpler versions of the generic equations. The arithmetic expressions of the equations are further simplified by:

- replacing the allocation variables a_c with 0 for regular components due to constraint 5 (see Section IV-D), and
- removing *min* functions when their second argument cannot exceed 1, for example when it consists of a single a_{c_i} element.

For instance, for an output port of a component c_2 that is only connected to an input port of a component c_3, the first *min* function used for the energy calculation (i.e., equation 3), is initially $min(1,(1-a_{c_3}))$ but is reduced to just $(1-a_{c_3})$ that corresponds to the connected c_3 component. A more detailed example that describes the calculation and simplification of a system properties, is presented in the next section.

V. Case study

In this section, we introduce the component-based system on which we apply the optimization model. The case study is the vision system of an underwater robot. The robot navigates autonomously underwater, fulfilling various missions such as tracking red buoys. As a hardware platform, the robot is equipped with an embedded board that contains a CPU and a GPU integrated on the same chip. The board is connected to a pair of camera sensors that provide a continuous flow of images of the surrounding environment, and to several actuators (i.e., thrusters) that allow underwater movement. The architecture of the vision system is constructed using the Rubus component model, as illustrated in Fig. 4.

The nine components of the vision system are divided in two trigger groups, i.e., group 1 containing six components (upper part of Fig. 4), and group 2 containing three components. The component functionalities are the following. Two components

TABLE III: The unrolled energy equations for the two groups of the vision system

$$
\begin{aligned}
&Energy_{\langle f_1,C_1\rangle} = \\
&\quad cpuEnergy(c_1)+ \\
&\quad cpuEnergy(c_2)+ \\
&\quad a_{c_3}*gpuEnergy(c_3)+(1-a_{c_3})*cpuEnergy(c_3)+ \\
&\quad a_{c_4}*gpuEnergy(c_4)+(1-a_{c_4})*cpuEnergy(c_4)+ \\
&\quad a_{c_5}*gpuEnergy(c_5)+(1-a_{c_5})*cpuEnergy(c_5)+ \\
&\quad cpuEnergy(c_6)+ \\
&\quad a_{c_3}*portSize(op_{c_1})*k_{eng}+ \\
&\quad a_{c_3}*portSize(op_{c_2})*k_{eng}+ \\
&\quad \big(a_{c_3}*min(1,1-a_{c_4}+1-a_{c_7})+(1-a_{c_3})*min(1,a_{c_4}+a_{c_7})\big)* \\
&\qquad portSize(op_{c_3})*k_{eng}+ \\
&\quad \big(a_{c_4}*(1-a_{c_5})+(1-a_{c_4})*a_{c_5}\big)*portSize(op_{c_4})*k_{eng}+ \\
&\quad \big(a_{c_5}+(1-a_{c_5})*a_{c_8}\big)*portSize(op_5)*k_{eng}
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
&Energy_{\langle f_2,C_2\rangle} = \\
&\quad a_{c_7}*gpuEnergy(c_7)+(1-a_{c_7})*cpuEnergy(c_7)+ \\
&\quad a_{c_8}*gpuEnergy(c_8)+(1-a_{c_8})*cpuEnergy(c_8)+ \\
&\quad cpuTime(c_9)+ \\
&\quad \big(a_{c_7}*(1-a_{c_8})+(1-a_{c_7})*a_{c_8}\big)*portSize(op_{c_7})*k_{eng}+ \\
&\quad a_{c_8}*portSize(op_{c_8})*k_{eng}
\end{aligned}
\tag{7}
$$

(i.e., *Camera1* and *Camera2*) are connected to the physical sensors, and convert the received raw data in readable frames. These frames are forwarded to the *MergeAndEnhance* component that reduces the noise and merges the frames. The output is converted into grayscale format by *ConvertGrayscale*. The *EdgeDetection* component filters the inputed frame and provides a black-and-white frame, where the white lines delimits the objects from the frame. There are two components (i.e., *CompressRGB* and *CompressGrayscale*) that compress system frames, while the *Logger* component stores them for logging purposes.

The system contains a total of five flexible components that are fit to be executed, if required, on GPU due to their functionality, i.e., processing images. For simplification, we numbered as c_1 to c_6 the components of group 1, from left to right. Similarly, the three components belonging to group 2 are numbered, from left to right, as c_7 to c_9.

The vision system properties calculated using the equations from Table II have a simplified form. For instance, Table III presents the unrolled energy usage calculation for both vision system groups. Equation 6 details the energy usage of group 1. The equation is not linear due to the fact that component c_3 (i.e., *MergeAndEnhance*) has an output port connected to two ports; this requires the *min* functions to remain. C_5 (i.e., *EdgeDetection*) is another component that also has an output port that communicates with two different components (i.e., c_6 and c_8). However, because c_6 (i.e., *ObjectDetection*) is a regular component (i.e., $a_{c_6} = 0$), the *min* function is removed from the equation.

For group 2, where each output port is single connected to a component, the energy usage reduces to equation 7. Furthermore, because c_9 (i.e., *Logger*) is a regular component (i.e., $a_{c_9} = 0$), this simplifies the equation even more. In this case, the equation is linear.

To compute allocation schemes, we use the CPLEX solver[2] developed by IBM. The unrolled system properties alongside with the constraints and optimization function of the vision system are converted into a CPLEX optimization model.

TABLE IV: The properties of the vision system components

Group	Component	Properties				
		Memory (bytes)			Time (msec)	
		CPU	GPU	GPUCPU	CPU	GPU
1	Camera1	56	0	0	3.2	0
1	Camera2	56	0	0	3.2	0
1	MergeAndEnhance	69	10575	24	4	0.6
1	ConvertGrayscale	60	8550	22	4	0.5
1	EdgeDetection	132	24750	22	1	0.6
1	ObjectDetection	124	0	0	6	0
2	CompressRGB	100	15300	22	3.1	0.5
2	CompressGrayscale	100	15300	22	3.2	0.5
2	Logger	70	0	0	4	0

GPU - The GPU memory requirement when allocated on the GPU
GPUCPU - The CPU memory requirement when allocated on the GPU

The requirements and characteristics of the vision system components are described in Table IV. As we do not have means for energy usage measurements, we use the literature results that show the GPU energy efficiency [7], and assume 2 energy usage units for each CPU-allocated component, and 1 energy usage unit for each GPU-allocated component. The flexible components and their characteristics are highlighted in gray. For example, the flexible component *MergeAndEnhance* requires 69 bytes of memory when allocated on the CPU, and delivers and execution time of 0.6 msec when allocated on the GPU. We mention that the time used to transfer a unit of data between distinct memory spaces (i.e., k_{tm}) is 0.00002 msec.

[2]https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer

Regarding the platform, the embedded board of the underwater robot is equipped with an AMD Accelerated Processing Unit[3], where the CPU and GPU are integrated on the same chip. The total memory of the chip is 200MB, but this memory should be used by the entire system of the robot. Therefore, only a part of the platform memory is available for the vision system. To make the allocation more interesting, we decided to use 350000 bytes as available CPU memory, and 500000 bytes as available GPU memory, respectively.

TABLE V: Four allocation scenarios for the vision system

Group	Component	Allocation			
		Fitness1	Fitness2	Fitness3	Fitness4
1	MergeAndEnhance	GPU	CPU	CPU	GPU
1	ConvertGrayscale	GPU	GPU	GPU	GPU
1	EdgeDetection	CPU	GPU	GPU	CPU
2	CompressRGB	GPU	GPU	GPU	GPU
2	CompressGrayscale	CPU	GPU	GPU	GPU

Fitness1 - Minimize the execution time of group 1
Fitness2 - Minimize the execution time of group 2
Fitness3 - Minimize all system properties (all weights are equal)
Fitness4 - Minimize all system properties (the CPU memory weight is double than all other weights)

Table V presents four allocation schemes of the vision system computed by the CPLEX solver when using a machine with a 2,6 GHz i7 processor and 16 GB of memory. For the *Fitness1* case, we focused on only minimizing the execution time of group 1 (i.e., $w_1^t = 1$, and all other weights set to 0). The result shows that two of the flexible components from group 1 are allocated on the GPU. The third flexible component, that has a lower execution time efficiency compared to the other two components, was allocated on the CPU, due to e.g., the memory limitation constraints. In the second case, we aim to instead minimize the execution time of group 2 (i.e., $w_2^t = 1$, and all other weights set to 0). The result shows the allocation on the GPU of the two flexible components of group 2. Furthermore, in this case, two of the flexible components of group 1 are allocated on the GPU due to e.g., memory limitation constraints. In *Fitness3* case, when all weights have equal non-zero values (i.e., 0.2), all flexible components, except *MergeAndEnhance*, are distributed over the GPU. In the last case, where w^c (i.e., CPU memory weigh parameter) has a double importance than the other weights, all flexible components, except *EdgeDetection*, are allocated on the GPU.

The time used by the solver to compute the allocation schemes is 0.01 seconds. To examine the calculation time for bigger systems, we constructed three scenarios described in Fig. 5. Scenario 1 consists of a chain of connected flexible components. While in Scenario 2, each flexible component communicates, via a single output port, with two other flexible components, in Scenario 3, each flexible component has two output ports connected to different components. In each scenario we vary the number of contained components, resulting

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

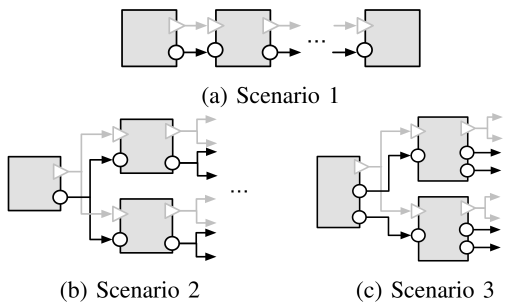

Fig. 5: Three types of scenarios for scalability evaluation

in three versions with 21, 31 and 41 components, respectively. In Scenario 2 and 3, in order for each component to have two output connections, an extra component is added, hence the even number of components. We added an extra component in Scenario 1 to have the same number of components in all three scenarios.

TABLE VI: Implementation and execution of the scenarios

	Number of				Average
	flexible components	output data ports	data connections	total operators	optimization time (ms)
Scenario 1	21	20	20	638	65
	31	30	30	938	142
	41	40	40	1273	3806
Scenario 2	21	20	38	883	146
	31	30	58	1344	395
	41	40	78	1784	(5918)
Scenario 3	21	38	38	1089	174
	31	58	58	1661	520
	41	78	78	2235	(10602)

Each version of the three scenarios was implemented in CPLEX by unrolling the general equations (see Table II). Table VI presents different information about the scenarios and their implementation, such as the total number of output data ports, data connections and implemented arithmetic operators. We provided random values between 1 and 99999 for the component properties and ran the optimization 1000 times for each scenario version, with different random property values each time. The last table column presents the average of the optimization time, using a quad-core 2.6Ghz machine.

Figure 6 gives a more detailed perspective of the time spent computing solutions for the considered scenarios. We notice from the figure and table information that the solver easily handles all the considered scenarios, in less than 30 seconds. Scenario 3 takes the longest time due to the fact that the equations have the highest number of operators. The experiment shows that the optimization time is mainly influenced by the number of flexible components. Other factors that have an impact over the optimization time are the number of output data ports and their distinct connections.

VI. Related Work

There is a huge amount of work on the software optimization subject such as the systematic literature review on the architecture optimization methods [8]. Among other aspects, the authors show that, from the total number of papers that study the optimization of component-based systems (i.e., 30 papers), only 13% (i.e., 4 papers) use exact optimization strategies (e.g., MINLP). Although exact methods provide optimal solutions, the difficulty of formally defining the allocation model, the search-space and the usually non-linearity (and computationally expensive) of the objective functions are major challenges in adopting them. We managed to formally define our optimization model, allowing us to use exact methods. Furthermore, we showed that our generic allocation model simplifies its search-space and complexity when applied on existing systems.

We mention the work of Seo et al. that focuses on the energy consumption estimation of component-based Java systems [9]. The work constructs a detailed optimization model of the system energy consumption. An interesting aspect is the energy usage of the communication when components reside in different Java Virtual Machines, on the same host. In our optimization model, the communication aspect is implicitly covered by the component's computational cost. We also treat the energy usage of components communication, but we specifically capture it in the optimization model.

Another work that deals with the energy usage is proposed by Goraczko et al. [10]. The authors develop an optimization model expressed using integer linear programming, that minimizes the system energy usage when the end-to-end time constraints are given. It is shown that (an older version of) the CPLEX solver calculates solutions on a dual-core 2GHz machine, in up to couple of minutes, for systems with more than 30 components. The model applies on embedded systems that have multi-CPUs. Similarly, our work focuses also on

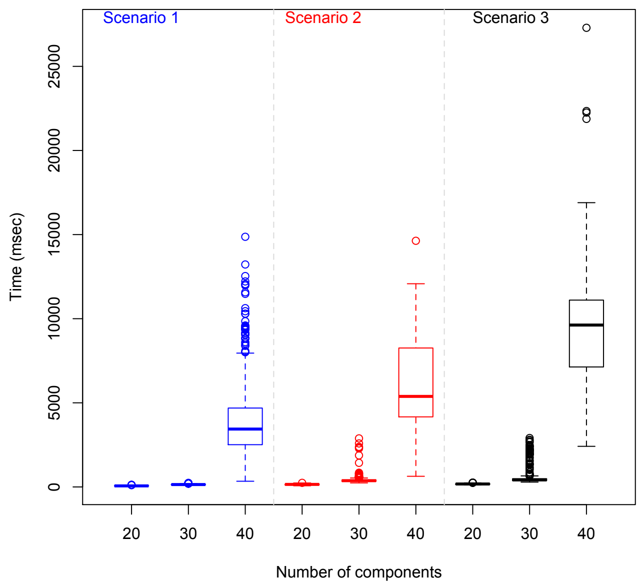

Fig. 6: The execution time used for computing solutions

embedded systems, but the processing units (i.e., CPU and GPU) have different characteristics. Therefore, the model of our system is more complex, containing the two distinct perspectives of the (CPU and GPU) processing units. Furthermore, when applying our optimization model on systems with e.g., 90 components, the newest version of the CPLEX solver computes solutions in a fast manner (i.e., 10 msec), when executed on a powerful machine (i.e., with four-core 2.6Ghz).

In the context of embedded real-time systems, Wang et al. provide a component-to-platform allocation model [11]. The proposed model considers the computation, communication, and memory resources of the components, and uses weights to define their importance in the allocation process. Interestingly, the components that require more resources have priority, being allocated first. In our work, all components have equal allocation priority w.r.t. their resource requirements. Furthermore, through the used flexible component concept, we increase the flexibility of the allocation regarding the component resource requirements. Similarly, we use the weight parameters to define the properties importance in the allocation process. The communication cost, represented in our work by adapters, is integrated in the way we calculate the system properties.

In our previous works, we have addressed the optimization challenge as follows. In one work, an optimization model is formally constructed to allocate component-based systems with GPUs [12]. This work presents its system model through a hypothetical component model and targets distributed systems. The model defines abstract system properties such as CPU capacity that describes the processor workload w.r.t. a conceptual reference unit. Another work that we also built on a hypothetical component model, allocates components during run-time on a distributed platform [13]. Similarly, this work formally describes the system using abstract properties such as CPU capacity, and covers the parallel allocation of components on GPU. Similarly to these previous two works, we formally define a component allocation for embedded systems with GPUs. In difference, the present work is constructed using an existing component model (i.e., Rubus) with its defined elements (e.g., flexible components, adapters). The previous works do not consider, as the present model does, the data transfer overhead between the CPU and GPU. Furthermore, we characterized the system using realistic properties. We do not explicitly cover parallel allocation of components on GPU as there are no defined mechanisms to support this feature in Rubus. Another difference is that we focus on the component allocation over compact platforms with single (CPU-GPU) processing chips.

Regarding the assumptions we made related to the energy usage of GPUs and CPUs, Huand et al. show the GPU efficiency over the CPU [7]. Using an existing (parallelizable) application, the authors compare the energy usage of a system when the application is executed by: *i)* the GPU, and *ii)* a single-thread CPU. It is showed that, to execute the same application, the GPU consumes 20 times less energy than the CPU. The energy calculations from the experiments are done using a power meter tool. Another aspect that helped us in constructing the reasoning related to the energy usage of components is that the energy consumption of a functionality that executes on GPU is not influenced by possible previous GPU executions. This leads to independent GPU energy usage of functionalities that are consecutively executed [14].

VII. Conclusions

In this work, we introduce an optimization allocation model for embedded systems with GPUs. The model optimizes important aspects of the embedded system domain, such as memory and energy usage. A novelty of the work is the formal description of the optimization model which allows to employ MINLP to compute, for given systems, optimal solutions (when they exist). The optimization model was applied on an existing component-based vision system showing the feasibility of the method. Furthermore, a scalability examination showed that a considered solver rapidly computes solutions (i.e., within 30 seconds) for systems with up to 40 components.

References

[1] "AUTOSAR - Technical Overview," http://www.autosar.org, accessed: 2018-01-22.

[2] 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.

[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] G. Campeanu, J. Carlson, and S. Sentilles, "Flexible components for development of embedded systems with GPUs," in *24th Asia-Pacific Software Engineering Conference*, December 2017.

[5] S. K. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," in *IEEE real-time and embedded technology and applications symposium*, 2004, pp. 536–543.

[6] T. Berthold, *Heuristic algorithms in global MINLP solvers*. Verlag Dr. Hut, 2014.

[7] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *Parallel & Distributed Processing, IEEE International Symposium on*. IEEE, 2009.

[8] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, 2013.

[9] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 421–424.

[10] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proceedings of the 45th annual design automation conference*. ACM, 2008, pp. 191–196.

[11] S. Wang, J. R. Merrick, and K. G. Shin, "Component allocation with multiple resource constraints for large embedded real-time software design," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*. IEEE, 2004.

[12] G. Campeanu, J. Carlson, and S. Sentilles, "Component allocation optimization for heterogeneous CPU-GPU embedded systems," in *The 40th Euromicro Conf. on Soft. Eng. and Advanced Applications*, 2014.

[13] G. Campeanu and M. Saadatmand, "Run-time component allocation in CPU-GPU embedded systems," in *Proceedings of the Symposium on Applied Computing*. ACM, 2017, pp. 1259–1265.

[14] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 2014, p. 28.