Mälardalen University Doctoral Dissertation 264

GPU Support for Component-based Development of Embedded Systems

Gabriel Campeanu

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Dissertations
No. 264

GPU SUPPORT FOR COMPONENT-BASED DEVELOPMENT OF EMBEDDED SYSTEMS

Gabriel Campeanu

2018

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISBN 978-91-7485-393-3
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 264

GPU SUPPORT FOR COMPONENT-BASED DEVELOPMENT OF EMBEDDED SYSTEMS

Gabriel Campeanu

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin
för innovation, design och teknik kommer att offentligen försvaras tisdagen
den 11 september 2018, 13.15 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Associate Professor Tomas Bures, Charles University in Prague

MÄLARDALEN UNIVERSITY
SWEDEN

Akademin för innovation, design och teknik

Abstract

One pressing challenge of many modern embedded systems is to successfully deal with the considerable amount of data that originates from the interaction with the environment. A recent solution comes from the use of GPUs. Equipped with a parallel execution model, the GPU excels in parallel processing applications, providing an improved performance compared to the CPU.

Another trend in the embedded systems domain is the use of component-based development. This software engineering paradigm that promotes construction of applications through the composition of software components, has been successfully used in the development of embedded systems. However, the existing approaches provide no specific support to develop embedded systems with GPUs. As a result, components with GPU capability need to encapsulate all the required GPU information in order to be successfully executed by the GPU. This leads to component specialization to specific platforms, hence drastically impeding component reusability.

Our main goal is to facilitate component-based development of embedded systems with GPUs. We introduce the concept of flexible component which increases the flexibility to design embedded systems with GPUs, by allowing the system developer to decided where to place the component, i.e., either on the CPU or GPU. Furthermore, we provide means to automatically generate the required information for flexible components corresponding to their hardware placement, and to improve component communication. Through the introduced support, components with GPU capability are platform-independent, being capable to be executed on a large variety of hardware (i.e., platforms with different GPU characteristics). Furthermore, an optimization step is introduced, which groups connected flexible components into single entities that behave as regular components. Dealing with components that can be executed either by the CPU or GPU, we also introduce an allocation optimization method. The proposed solution, implemented using a mathematical solver, offers alternative options in optimizing particular system goals (e.g., memory and energy usage).

ISBN 978-91-7485-393-3
ISSN 1651-4238

Abstract

One pressing challenge of many modern embedded systems is to successfully deal with the considerable amount of data that originates from the interaction with the environment. A recent solution comes from the use of GPUs. Equipped with a parallel execution model, the GPU excels in parallel processing applications, providing an improved performance compared to the CPU.

Another trend in the embedded systems domain is the use of component-based development. This software engineering paradigm that promotes construction of applications through the composition of software components, has been successfully used in the development of embedded systems. However, the existing approaches provide no specific support to develop embedded systems with GPUs. As a result, components with GPU capability need to encapsulate all the required GPU information in order to be successfully executed by the GPU. This leads to component specialization to specific platforms, hence drastically impeding component reusability.

Our main goal is to facilitate component-based development of embedded systems with GPUs. We introduce the concept of flexible component which increases the flexibility to design embedded systems with GPUs, by allowing the system developer to decide where to place the component, i.e., either on the CPU or GPU. Furthermore, we provide means to automatically generate the required information for flexible components corresponding to their hardware placement, and to improve component communication. Through the introduced support, components with GPU capability are platform-independent, being capable to be executed on a large variety of hardware (i.e., platforms with different GPU characteristics). Furthermore, an optimization step is introduced, which groups connected flexible components into single entities that behave as regular components. Dealing with components that can be executed either by the CPU or GPU, we also introduce an allocation optimization method. The proposed solution, implemented using a mathematical solver, offers alternative options in optimizing particular system goals (e.g., minimize memory and energy usage).

Acknowledgment

This journey, my PhD studies, would not have been possible without the help and support of my supervisors, Jan Carlson, Séverine Sentilles and Ivica Crnković. Even now I remember that, during the first supervisory meeting, Ivica said that the 5 years of the PhD is a short period and it will go so fast. Today, after 6 years, I understand what he meant and many things that I wanted to do remained untouched due to the lack of time. Still, I am happy with my achievements and I want to thank you, my supervisors, for giving me the chance to embark on this amazing adventure that shaped me into a more mature and experienced person.

There were many ups and downs during the PhD studies, and, besides my supervisors, my friends that we shared the same office, offered me a great support. Julieth, Anita, Irfan, Omar, Husni and Filip, thank you for the great moments and discussions that were making the office such a pleasant environment.

My appreciations go to my wonderful colleagues and friends from IDT. You made my life so much fun and created such a great work environment.

Last but not least, I want to express my gratitude to some special persons. To my wife, Cristina, thank you for cheering me up so many times, and for the support you offered during these last and most difficult years of my PhD. To my family – mum, dad, sis – thank you for the unlimited encourages.

Gabriel Campeanu
Västerås, August, 2018

List of publications

Key peer-reviewed publications related to the thesis

Paper A: *Allocation Optimization of Component-based Embedded Systems with GPUs* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, The 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018.

Paper B: *Optimized Realization of Software Components with Flexible OpenCL Functionality* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, The 13th International Conference of Evaluation of Novel Approaches to Software Engineering, ENASE 2018.
Best student paper award.

Paper C: *Flexible Components for Development of Embedded Systems with GPUs* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, The 24th Asia-Pacific Software Engineering Conference, APSEC 2017.

Paper D: *Developing CPU-GPU Embedded Systems using Platform-Agnostic Components* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, The 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017.

Paper E: *Extending the Rubus Component Model with GPU-aware components* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, Saad Mubeen, In

Proceeding of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2016.

Paper F: *A GPU-aware Component Model Extension for Heterogeneous Embedded Systems* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, In Proceedings of the 10th International Conference on Software Engineering Advances, ICSEA 2015.
Best paper award.

Additional peer-reviewed publications related to the thesis

- *Scavenging Run-time Resources to Boost Utilization in Component-based Embedded Systems with GPUs* – Gabriel Campeanu, Saad Mubeen, The International Journal on Advances in Software, IARIA JAS 2018.
- *Facilitating Component Reusability in Embedded Systems with GPUs* – Gabriel Campeanu, The 16th International Conference on Software Engineering and Formal Methods, SEFM 2018.
- *Improving Run-Time Memory Utilization of Component-based Embedded Systems with Non-Critical Functionality* – Gabriel Campeanu, Saad Mubeen, The Twelfth International Conference on Software Engineering Advances, ICSEA 2017.
- *Parallel Execution Optimization of GPU-aware Components in Embedded Systems* – Gabriel Campeanu, The 29th International Conference on Software Engineering & Knowledge Engineering, SEKE 2017.
- *Run-Time Component Allocation in CPU-GPU Embedded Systems* – Gabriel Campeanu, Mehrdad Saadatmand, The 32nd ACM SIGAPP Symposium On Applied Computing, SAC 2017.

- *A 2-Layer Component-based Architecture for Heterogeneous CPU-GPU Embedded Systems* – Gabriel Campeanu, Mehrdad Saadatmand, The 13th International Conference on Information Technology: New Generations, ITNG 2016.

- *Component Allocation Optimization for Heterogeneous CPU-GPU Embedded Systems* – Gabriel Campeanu, Jan Carlson, Séverine Sentilles, The 40th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2014.

Other publications

- *A Mapping Study on Microservice Architectures of Internet of Things and Cloud Computing Solutions* – Gabriel Campeanu, The 7th Mediterranean Conference on Embedded Computing, MECO 2018.

- *Support for High Performance Using Heterogeneous Embedded Systems – a Ph.D. Research Proposal* – Gabriel Campeanu, The 18th International Doctoral Symposium on Components and Architecture, WCOP 2013.

- *The Black Pearl: An Autonomous Underwater Vehicle* – Carl Ahlberg, Lars Asplund, Gabriel Campeanu, Federico Ciccozzi, Fredrik Ekstrand, Mikael Ekström, Juraj Feljan, Andreas Gustavsson, Séverine Sentilles, Ivan Svogor, Emil Segerblad, Technical report.

Contents

I

Chapter 1

Introduction

Nowadays, computation devices are becoming increasingly common in systems from almost all domains. The devices are embedded in the systems and refereed as *embedded systems*. For example, we mention pacemakers from medicine, satellite navigation systems from avionics, and cruise control systems from the automotive domain. The embedded industry, valued to over USD 144 billion in 2014, is in a growing pace and will reach, in 2023, to a market size of almost USD 240 billion [1]. In fact, 98% of all processors that are produced worldwide, are used in embedded systems [2].

Many of the modern embedded systems deal with a huge amount of information resulted from the interaction with the environment. This data needs to be processed with a sufficient performance in order for the system to handle, in real-time, the environment changes. For instance, the autonomous Google car[1] receives 750 MB of data per second from its sensors (e.g., cameras, LIDAR). This data requires to be processed with a sufficient performance in order to e.g., detect moving pedestrians and vehicles.

One trend in embedded systems is the usage of boards with Graphics Processing Units (GPUs). Equipped with thousands of computation threads, the GPU provides an improved performance compared to the CPU, in the context

[1]https://waymo.com/

of data-parallel applications, where each thread executes the same instructions on different data. Today, various vendors merge together the CPU and GPU onto the same physical board. This allows an overall improvement of the system performance when a specific workload is distributed on the appropriate processing unit, such as sequential computation on the CPU and parallel computation on GPU. On the market, there exist many types of boards with GPUs, with different characteristics (e.g., size, computation resources), which make them suitable for specific applications. For example, there are boards with GPUs that have high computation power used in high-performance military systems, but also GPUs with lower computation power utilized in smart wristwatches.

Another trend in embedded systems is the usage of component-based development (CBD) [3]. This software engineering paradigm promotes the construction of systems through composition of already existing software units called *software components*. The advantages that come with the usage of CBD include an increased productivity, efficiency and a shorter time-to-market. CBD proved to be a successful solution in development of industrial embedded systems, through the usage of component models such as AUTOSAR [4], IEC 61131 [5] or Rubus [6].

1.1 Problem statement and research goals

The existing component models used in the development of embedded applications offer no specific GPU support. Several disadvantages are introduced by using these existing approaches, which diminish the benefits of developing embedded systems with GPUs, by using CBD. In the following paragraphs, we introduce the shortcomings addressed by the thesis.

In the context of embedded systems with GPUs, the component developer would explicitly construct components with functionality to be executed by specific processing units, i.e., either the CPU or GPU. The system developer, when constructing the component-based application, is restricted to use certain components in order to conform with the platform characteristics. For example, for a platform that does not contain GPU, the system developer is restricted from using components with GPU capability. Moreover, this limitation is increased by the fact that sometimes, the detailed platform characteristics are unknown at the system design-time. For example, without knowing, during system design-time, the detailed characteristics of the GPU, the system developer is restricted from using components with GPU capability, that have high GPU resource requirements.

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

There is another shortcoming related to the development of a component with GPU capability. The component developer is responsible, besides constructing the actual component functionality, to address specific information to access and use the GPU. For example, the component developer needs to specify the number of GPU threads used to execute the component functionality. This leads to a complex and error-prone development.

Once the component-based application is constructed, the allocation of the components to the hardware is an important step in the development of embedded systems with GPUs. The heterogeneity of the hardware (i.e., platforms with CPUs and GPUs) and the fact that the application contains components with different (CPU and GPU) characteristics, introduces a challenge of how to allocate the system functionality in order to utilize the hardware resources in the best way.

Considering the previously described shortcomings of the CBD in embedded systems with GPUs, we state the overall goal of this thesis:

To introduce specific GPU support in component-based development in order to facilitate the construction of embedded systems with GPUs.

More specifically, the thesis aims to: *i)* introduce theoretical concepts to tackle the existing CBD shortcomings, and *ii)* to show the feasibility of the introduced concepts.

The overall thesis goal is quite broad and addresses many facets of component-based development of embedded systems with GPUs. Therefore, for the work of this thesis, we refine it into three research (sub-)goals (RGs). The objectives of these goals are to explore the existing GPU-aware support regarding component-based development, to facilitate CBD of embedded systems with GPUs via particular GPU-aware mechanisms, and to assist the component-to-hardware allocation challenge. The specific research goals addressed by this thesis are defined as follows:

RG1: *Describe the existing research that targets GPU support in systems that follow a component-based approach.*

RG2: *Introduce mechanisms to component models for embedded systems, in order to facilitate the construction of applications with GPU capability.*

RG3: *Automatically determine suitable allocations for components with GPU capabilities.*

The starting step to address the main goal is: *i)* to explore and describe the existing needs of modern systems for embracing GPUs, and *ii)* the existing GPU-aware support provided by component-based development. We specifically pay attention to the embedded systems domain and how, if any, component models address the GPU-aware development.

Using the knowledge obtained from the first goal, a natural continuation in addressing the main goal is to assist in the development of embedded systems with GPUs. RG2 targets the required GPU-aware mechanisms to ease CBD for embedded systems. We mention here concepts to facilitate the GPU resources access such as memory and computation threads.

The last research goal aims to ease the construction of embedded systems with GPUs by introducing (automated) means to handle the functionality allocation onto the physical platform. Indeed, providing allocation solutions may be a challenge in the context of embedded systems with GPUs. On one side we have the software application which is composed of components that have strict requirements for the CPU or GPU resource utilization, and on the other side, the platform has physical limitations with respect to the available resources.

1.2 Contributions

This section describes the main contributions of this thesis. There are four contributions, described in the following paragraphs. These contributions address the overall thesis goal and the three specific goals introduced in the previous sections. While studying the state-of-the-art of component-based construction of applications with GPU capability, the lack of solutions in the domain targeted by our work has been identified. This has led to contribution 1. We have introduced our own specific solutions to facilitate the component-based development of embedded systems with GPUs. The theoretical solutions belong to contribution 2, while their practical realization are represented by contribution 3. The last contribution introduces (automatic) means to address functionality-to-hardware allocation when constructing embedded systems with GPUs. The four contributions are the following:

Contribution 1. *A description of the scientific research regarding GPU support in component-based systems.*

With this contribution, we review the state-of-the-art and describe the on-going research regarding GPU support in component-based systems. More specifically, we looked at: *i)* the trends of the research studies that target component-based systems with GPUs, and *ii)* the specific solutions used by these studies.

The research trends show that, up to 2009, there is no particular interest in the component-based applications with GPU capability. The increased interest may have been triggered by the fact that, from 2009, several (software and hardware) GPU technologies were released. Another aspect captured by the trends is that most of the research is done in academia. The second part of this contribution reveals that most of the studies do not use specific component models to target systems with GPUs. Various mechanisms are used to handle the GPU aspects, where the programming and modeling are the most utilized ones.

Contribution 1 is covered by Chapter 3 via a systematic literature review.

Contribution 2. *Mechanisms that specifically introduce GPU support for pipe-and-filter component models.*

This contribution targets RG2 and has the purpose to introduce theoretical concepts to facilitate component-based development of embedded systems. The contribution focuses on *pipe-and-filter* component models, and introduces the following concepts:

- the flexible component,
- a way to optimize groups of flexible components, and
- the component communication support.

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

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

Due to the different characteristics of embedded platforms with GPUs, components with GPU capability require different activities (corresponding to the platform characteristics) for data communication. We improve the component communication via special artifacts called *adapters*. Depending on the platform characteristics, the appropriate adapters are automatically introduce to facilitate the component communication.

Contribution 2, which is the core part of the thesis, has been published in Papers B, C, D, E and F. A comprehensive description of it is given in Chapter 4.

Contribution 3. *An extension of the Rubus component model to implement the introduced mechanisms.*

This contribution targets RG2 and presents a way to implement the introduced theoretical concepts. The realization is done using the Rubus component model, briefly presented as follows.

The flexible components are optimized into flexible groups which are translated, through a number of transformation and code generation steps, into regular Rubus components. The resulted components are automatically populated with the required platform information in order to be executed on the selected platform (i.e., CPU or GPU). The configuration interface of a flexible component which is inherited by a flexible group is realized as a regular component port, in order to not introduce additional component model elements.

The adapters, facilitating the communication between the resulted regular components, are realized as regular Rubus components with a single input data port and a single output data port.

Contribution 3, which complements the theoretical concepts of contribution 2 with their implementation, has been published in Papers B, C, D, E and F. A detailed description that contains code snippets of the concepts implementation is found in Chapter 4.

Contribution 4. *An allocation method that automatically finds suitable component allocations for embedded systems with GPUs.*

The concept of flexible component presented by contribution 2, introduces a challenge regarding the flexible component-to-hardware allocation. On one side, the application is composed of flexible components with different CPU and GPU resource requirements, while on the other side, the platform has physical limitations regarding the CPU and GPU resources. Deciding the allocation of the flexible components while considering important aspects of the embedded systems domain (e.g., memory and energy usage) is facilitated through an automatic method. Using exact optimization methods (i.e., mixed-integer non linear programming), the contribution delivers optimal solutions (if exist), with respect with the decided optimization criteria.

Contribution 4 has been published in Paper A. The description that contains the mathematical formulation, its implementation and evaluation, is enclosed in Chapter 5.

1.3 Research process

A series of guidelines regarding research methodology in software engineering is provided by Basili [7]. The engineering method introduced by Basili is to "*observe existing solutions, propose better solutions, build/develop, measure and analyze, and repeat the process*". Following his method, we derive our research process as illustrated by Figure 1.1.

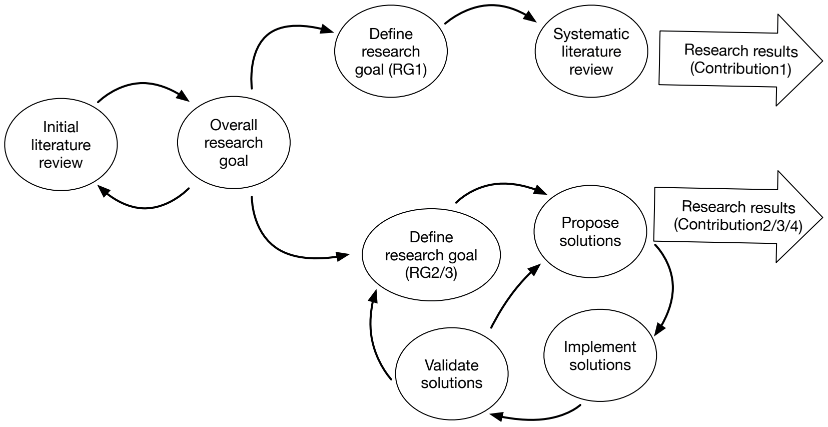

Initial literature review
Overall research goal
Define research goal (RG1)
Systematic literature review
Research results (Contribution1)
Define research goal (RG2/3)
Propose solutions
Research results (Contribution2/3/4)
Implement solutions
Validate solutions

Figure 1.1: Overview of the used research process

We started with an *initial literature review*, where we looked into the state-of-the-art knowledge regarding the component-based development of embedded systems with GPUs. The review showed that there is no specific support for GPU development in this domain, which lead us to the *overall research goal*, i.e., to facilitate component-based development of embedded systems with GPUs. After setting the thesis goal, we went into more depth, by defining the research details, as follows.

Facilitating the component-based development of embedded systems with GPUs is a broad problem, therefore we defined research goals that are more specific. The first goal (RG1) is to present the existing knowledge regarding the development of systems with GPU capability, using a component-based approach. The process of RG1 is presented in the upper part of Figure 1.1, where, after defining the research goal, we carry out a systematic literature review (SLR) using well-established guidelines [8]. The output of this SLR (i.e., Contribution 1) is a research result of this thesis.

For the rest of the research goals (i.e., RG2 and RG3) we used the iterative process presented in the bottom part of Figure 1.1, based on the method described by Basili [7]. After defining the second research goal (i.e., RG2), that is to facilitate, via specific mechanisms, the construction of embedded systems with GPUs, we *propose solutions* that are *implemented* as an extension of an existing component model. The extended component model is *validated* through a case study. During this part of the process, there is a reverse step from validation to solution proposal, given that the theoretical proposed solutions may be changed by the practical side of the validation. The results (i.e., Contribution 2 and 3) of this research goal are the core contributions of the thesis.

The third research goal (i.e., RG3), that is, to provide methods for component-to-hardware allocation, follow the same process as RG2. More specifically, after defining the research goal, we propose a formal allocation method that is implemented using an existing solver. Finally, the allocation method is validated using a case study. The allocation method (i.e., Contribution 4) represent the final contribution of this thesis.

1.4 Thesis outline

The thesis contains the following seven chapters:

Chapter 1: Introduction contains an overview of the research context, introducing the motivation of the work. Additionally, the thesis problem is stated and the goals are set. In the last part, the research process and a description of the contributions are introduced.

Chapter 2: Background introduces details about the context of the work, i.e., a description of embedded systems, the component-based development methodology, and GPUs and the development of GPU applications.

Chapter 3: GPU support in component-based systems is a systematic literature review that describes the trends and the detailed solutions of research studies on component-based systems with GPUs.

Chapter 4: GPU-aware mechanisms starts by describing the existing challenges addressed by the thesis and our view on a development process overview which contains concepts that tackle the presented challenges. The chapter continues by providing a detailed description of the concepts that we use to facilitate component-based development of embedded systems with GPUs. The presented concepts are evaluated using a vision system of an underwater robot case study.

Chapter 5: Allocation optimization presents a methods that automatically provides optimized solutions regarding the component-to-hardware allocation. The method is evaluated using the same vision system case study.

Chapter 6: Related work examines our introduced concepts in relation to existing work.

Chapter 7: Conclusions and future work presents the conclusions of this thesis and describes possible directions for its continuation.

II

Chapter 2

Background

The thesis main goal is to facilitate CBD for embedded systems with GPUs. In this chapter we introduce background information about the context of this work. In particular, we start by introducing embedded systems, followed by the component-based development methodology. In the last part of the chapter, we focus on the GPU particularities and how to program it using the OpenCL environment.

2.1 Embedded systems

Nowadays, computer systems are part of a majority of all developed electronic products. In general, there are differences between general-purpose and specific computer systems. The general-purpose systems such as personal computers, are used in various general-computing activities such as emailing, Internet surfing and office applications. In this thesis, the focus is on the specific-type of systems, that is on the embedded systems; these systems have specialized purposes. Examples of embedded systems range from simple devices such as microwaves ovens or music-players, to complex systems such as airplanes or factories. One of the definitions used to describe an embedded system is provided by Bar and Massa, as follows:

Definition 1. *"An embedded system is a combination of computer hardware and software – and perhaps additional parts, either mechanical or electronic – design to perform a dedicated function"* [9].

A typical embedded system is characterized by limited size, memory and processing power, and a need for low power consumption. For example, while a general-purpose system may be equipped with several gigabytes of RAM memory, an embedded systems has a limit of e.g., few megabytes or kilobytes of memory system. Besides these typical characteristics, there are specialized embedded systems with more stringent physical requirements that are employed in specific domains. For instance, rugged embedded systems are those systems that operate in harsh environment conditions such as extreme temperatures or wet conditions [10]. Satellites are such embedded systems; they are built to endure extreme temperatures out in space and are resistant to radiations. Another specific characteristic of embedded systems is the real-time requirements that some applications may be subject to. A real-time embedded system guarantees to deliver a response within a well defined period of time. The front-airbag of a car is an example of such embedded system; the trigger to deflates it is initiated at about 25 milliseconds after the crash.

Other properties (also know as extra-functional properties) are important for the embedded system domain, such as performance and reliability [11]. For example, the embedded systems used in mobile computing domain are characterized by high performance [12]. When constructing embedded systems, the focus is not only concentrated on the software development activity, but also on addressing the extra-functional properties. The extra-functional properties cover a large diversity of features such as performance, availability, security and maintainability. Some of the properties may have various facets. For instance, performance includes aspects such as power consumption but also time-related characteristics such as execution time and response time.

2.2 Component-based development

In the last two decades, software applications have greatly increased in size and complexity [13]. Software development methods utilized in developing applications faced new challenges in efficiently addressing the increased extra-function properties (e.g., maintainability, performance). A feasible solution to tackle these challenges is component-based development (CBD). Its objective is to address the software applications complexity by composing software blocks called *software components*. In this way, complex applications can be easily developed by composing components.

A definition of the software component is provided by Szyperski as follows:

Definition 2. *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"* [14].

With his definition, Szyperski introduces several characteristics of a software component such as *interface* and *composition*. An *interface*, used to enable interaction between components, is a specification of the component access point. There are several types of interfaces such as operation-based and port-based [15]. The so-called *port-based* interfaces, used in our work, are entries for sending/receiving different data types between components. Composition describes the rules and mechanisms used to combine components. The component may be developed by an external software producer so called third-party, and used without any knowledge of how the component was created. Ideally, all components should be available on a market, as commercial-off-the-shelf (COTS) components, from where any user or company can use and reuse components according to their needs. Among benefits of employing CBD when developing systems, we mention the ability to reuse the same component developed either in-house or by third-parties, thus improving the development efficiency.

An important concept in the CBD community is the notion of the component model. The component model defines standards for: *i)* building individual software components; and *ii)* assembling components into systems. For example, the Microsoft's Component Object Model (COM) [16] enforces that all components should be constructed with a *IUnknown* interface. There exists a big variety of component models, some that are focused on specific application domains (e.g., embedded systems for automotive industry) and other build on specific technological platforms (e.g., Enterprise Java Beans).

CBD is successfully used in building complex desktop applications through general-purpose component models such as CORBA [17], .NET [18], COM [16] and JavaBeans [19]. When it comes to embedded systems, the general-purpose component models lack means to handle the specifics of this domain such as real-time properties and low resource utilization [15]. For example, while a general-purpose system may be equipped with several gigabytes of RAM memory, an embedded systems has a limit of e.g., few megabytes or kilobytes of memory system. Another specific characteristic of embedded systems is the real-time requirements that some applications may be subject to. A real-time embedded system guarantees to deliver a response within a well defined period of time. However, several dedicated component models manage to provide feasible solution in developing embedded systems applications. For example, in the automotive industry, the AUTOSAR framework [4] is used as a standard of automotive development.

Many component models used in different embedded systems domains are constructed following well-known architectural styles [15]. These styles describe e.g., constraints on how components can be combined. In general, different architectural styles employ specific interaction styles. For example, the client-server architectural style that may be adopted in a distributed embedded system, specifies a component that sends a request for some data while another connected component responds to the request. In this particular style, the way the components communicate with each other is known as the *request-response* interaction style. Other interaction styles include the request-response, pipe-and-filter, broadcast, blackboard and publish-subscribe styles [15].

The work of this thesis focuses on component models that utilize a *pipe-and-filter* interaction style. In this context, components that process data behave as *filters* while the connections between components are seen as *pipes* that transfers data from one component to another. The reason of employing such a pipe-and-filter-based component model in embedded systems is because it provides a sufficient predictability level with respect to analysis of temporal behavior required to satisfy the real-time specifications of an embedded system. A pipe-and-filter component model is based on the control flow paradigm, where the control of the system at a specific time is owned by a single component and is passed to other components through specific mechanisms. Another characteristic of this style is the the unidirectional character of the component communication. Furthermore, in some component models, there is a distinct separation between data and control flow.

Among the component models that follow the pipe-and-filter style we mention ProCom [20], COMDES II [21] used in academia and IEC 61131 [22] and Rubus [6] employed by industry. These component models may be applied to various embedded system areas, such as automotive (addressed by Rubus) and industrial programmable controllers (addressed by IEC 61131). Our work focuses on the embedded systems domain that deal with large amount of data that can benefit from using GPU usage. Moreover, the embedded systems that we target can be addressed by using pipe-and-filter-based component models. A good example is the automotive industry where the software applications used by Volvo construction equipment vehicles (e.g., excavators) are developed using the Rubus component model and one of the current direction is to make them autonomous [23].

2.2.1 The Rubus component model

A part of our work focuses on extending the Rubus component model with GPU awareness. Therefore, the following paragraph describes the Rubus components and the component communication mechanism. The Rubus component model follows the pipe-and-filter interaction style, and has a separation between data and control flow. Every Rubus component is equipped with two types of ports, i.e., data and trigger ports. Through the trigger ports, the control is passed between components; similarly, data is passed using the data ports. A Rubus component is equipped with a single input trigger port and a single output trigger port; regarding data ports, a component may have one or several (input and output) ports.

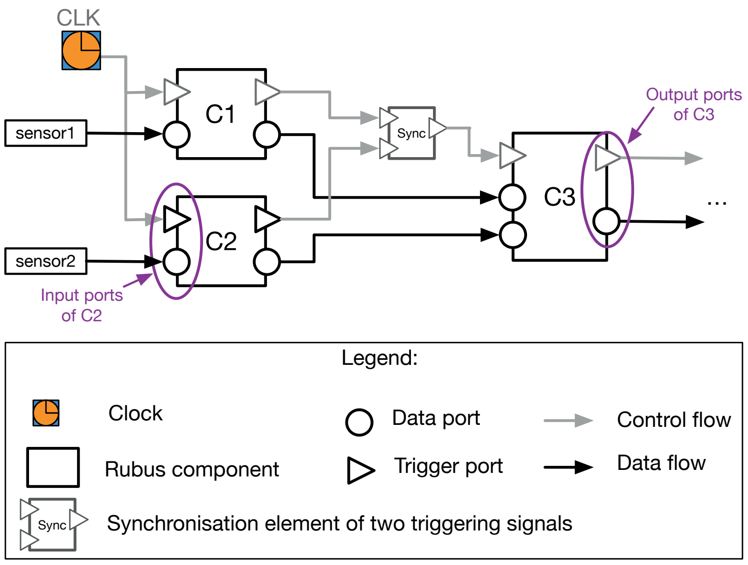


Figure 2.1: Connected Rubus components

Figure 2.1 presents a Rubus (sub-)system composed of three connected components, i.e., $C1$, $C2$ and $C3$. At a periodic interval of time specified by the clock element *CLK*, component $C1$ is triggered through its trigger input port, i.e., it receives the control to execute its behavior. The execution semantic of the Rubus component is Read-Execute-Write. It means that $C1$ was in an inactive mode before being triggered by the clock element. Once activated, the component switches to Read mode where it reads the data from its input data port, received from *sensor1*. During Execute mode, the component performs its functionality using the input data. After the execution completion, the result is written in the output data port during Write mode, and the output trigger port is activated. The control is passed to the next connected component (i.e., $C3$) through the output trigger port, and $C1$ returns to the inactive state. We notice that there are two components triggered by the same clock. The order of their execution is e.g., based on the component priorities or the scheduling policy of the OS.

2.3 Graphics Processing Units

Initially when GPUs appeared in the late 90s, they were only used for graphics-based applications, excelling in rendering high-definition graphics scenes. Over time, GPUs were equipped with an increased computation capability, and became easier to program. Having now means to easily program GPUs, developers manage to port many non-graphical computationally demanding applications to the GPUs, and were referred as General-Purpose GPUs [24]. For instance, cryptography applications [25] and Monte Carlo simulations [26] have GPU-based solutions.

GPUs, through their massive parallel processing capabilities, manage to outperform the traditional sequential-based CPUs in heavy data-parallel computations. For example, the bio-molecular simulations achieved a 20 times speed-up when executed on GPUs [27].

CPUs and GPUs are constructed with different architecture structures, as follows. Designed as a general-purpose unit to handle any computation task, the CPU is optimized for lower operation latency (by using large cache memories). It may consist of one or several processing cores and can handle few software threads. On the other hand, the GPU is built as a special-purpose unit, being specialized in highly parallel computations. It is constructed with tens of processing cores that can handle thousands of computation threads.

Various vendors such as Intel, AMD, NVIDIA, Altera, IBM, Samsung and Xilinx develop embedded-board platforms with GPUs. The GPU is made part of these platforms in two ways, either as a discrete unit or integrated into the platform. When the GPU is discrete (referred to as *dGPU*), it has its own private memory. For instance, the Condor GR2[1] is a discrete GPU that is used in embedded systems. When the GPU is integrated (known as *iGPU*) on the same chip with the CPU, the memory is shared between the CPU and GPU. For example, AMD Kabini[2] is a chip-set that contains an AMD CPU and GPU integrated together.

Embedded boards with iGPU architectures are the predominant platforms used in industry due to their lower cost, size and energy usage. For instance, we mention here wearable devices such as Cronovo smart-watch[3]. On the other side, dGPUs, with larger physical size and increased GPU resources, are used by systems that require higher performance. We mention systems from aerospace and defense domains using ruggedized VPX3U GPU[4].

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

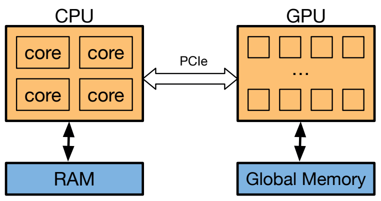
Figure 2.2 illustrates the architectures of different platforms with GPUs. Systems with dGPUs (Fig. 2.2(a)) are characterized by distinct memory systems, where data needs to be transfered from one system to the other via e.g., a PCIexpress bus. Most platforms with iGPUs have the same physical memories divided into distinct parts, i.e., one for the CPU and the other for the GPU (Fig. 2.2(b)). In this case, there is still need for data transfer activities with a minimized transfer overhead due to the physical location of the data (i.e., on the same memory chip). There are improved platforms with an optimized memory access which offer a shared virtual memory (SVM) space (Fig. 2.2(c)). To

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

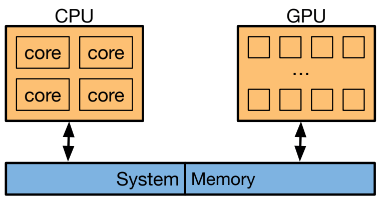
[2]http://www.amd.com/en-us/products/processors/desktop/athlon

[3]http://www.cronovo.com

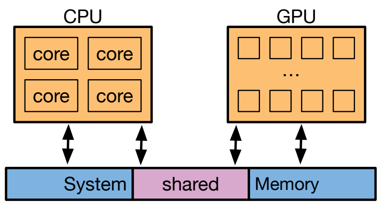
[4]https://wolfadvancedtechnology.com/products/vpx3u-tesla-m6/



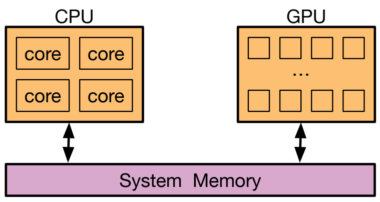
(a) dGPU system with distinct memory systems



(b) iGPU system with distinct memory addresses



(c) iGPU system with partially-shared memory addresses



(d) iGPU system with full shared memory addresses

Figure 2.2: Embedded platforms with different GPU architectures

place data on SVM, specific transfer activities are used; on the other hand, no specific activities are used by either the CPU or GPU to access the data from SVM. The latest and most technological advanced architecture (Fig. 2.2(d)) offers simultaneous access to the same memory for both CPU and GPU, without any need for data transfer.

2.3.1 Development of GPU applications

The challenge of leveraging the parallel computing engine of GPUs and developing software applications that transparently scales their parallelism to the GPUs' many-cores, was tackled by several GPU programming models. The two most popular programming models are CUDA [28] and OpenCL [29]. While CUDA was developed by NVIDIA to address only NVIDIA GPUs, OpenCL is a general model supported by multiple platforms and vendors (e.g., Intel, AMD, NVIDIA, Altera, IBM, Samsung, Xilinx), that targets various processing units, including CPUs and GPUs. Basically, both programming models have the same concepts utilized through different terms. We utilized in our work the OpenCL programming model to develop the GPU functionality.

While using OpenCL to develop an application, there are several hierarchical steps that needs to be respected. We describe these steps using a simple application example, i.e., the multiplication of two vectors. The steps are the following:

1. Defining the platform

   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 selected device (e.g., iGPU) are sent using a *command queue* mechanism.

   Listing 2.1 presents the required steps for constructing the environment for the vector multiplication application. It starts by creating a *platform* (line 7), selecting a GPU *device* to be used (line 10), defining a *context* that contains the GPU device (line 14), and finally, creating a *command queue* (line 17) through which commands are sent to the GPU.

   Listing 2.1: Setting up the GPU environment

```
1  cl_platform_id platform_id = NULL;
2  cl_device_id device_id = NULL;
3  cl_uint ret_num_devices;
4  cl_uint ret_num_platforms;
5
6  //create a platform
7  clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
8
9  //define the GPU compute device to be used
10 clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &
         ret_num_devices);
11
12
```

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

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

2. Creating and build the program

   A *program* to hold the defined kernel is created and compiled. Listing 2.2 presents the creation of the program which contains the kernel function *vec_mult* (line 2) and its compilation (line 5).

Listing 2.2: Creating and building the *program* setting

```
1  // Create a program from the kernel source
2  cl_program program = clCreateProgramWithSource(context, 1, (const
      char **)&vec_mult, NULL , NULL);
3
4  // Build the program
5  clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

3. Creating memory objects

   A next step is the allocation of the memory *buffers* on the device to hold data. For platforms with full shared memory, this step is skipped. In this example, we assume that the platform has distinct memory addresses, one for the CPU and another for the GPU. Listing 2.3 presents the allocation on the device of two memory buffers to hold the input data (line 1 and 2), and one memory buffer to retain the multiplication result (line 3).

Listing 2.3: Allocation of memory buffers

```
1  a_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) *
     n, NULL, NULL);
2  b_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) *
     n, NULL, NULL);
3  c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float)
     * n, NULL, NULL);
```

4. Defining the kernel

   The functionality, also known as the *kernel*, contains a function header and a function body. The function header contains the name of the function and its parameters. We mention that the qualifier *__kernel* declares the function to be a kernel function.

   For our application example, Listing 2.4 presents the kernel definition, where the function header is presented at line 1. The name of the kernel function is *vec_mult* followed by its input and output parameters. The body of the kernel (line 3, 4 and 5) describes the kernel functionality as follows. In line 3, the *id* of each used GPU thread is computed, while in line 5, the GPU thread executes the multiplication operation. In line 4, we make sure we do not exceed the vectors' length.

   Listing 2.4: The creation of a *kernel* object

```
1  __kernel void vec_mult (__global const float *input_a, __global
        const float *input_b, __global float *output_c, __global
        const int *n)
2  {
3   int id = get_global_id(0);
4   if (x >= n) return;
5   c[id] = a[id] * b[id];
6  }
```

   We mention that the kernel is not necessary to be defined at this step and it may be defined even before setting the platform (step 1).

   Once the kernel function is defined, a kernel object is created and arguments are attached to it. Listing 2.5 describes the creation of a kernel object (line 1) and the setting up of its four parameters (line 3-6).

   Listing 2.5: The kernel code

```
1  cl_kernel kernel = clCreateKernel(program, "vec_mult", NULL);
2
3  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_in);
4  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_in);
5  clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_out);
6  clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
```

5. Submitting commands

   In this step, various commands are issued such as data transfer and kernel execution commands. In Listing 2.6, we start by transferring the input data from the host to the device memory (lines 1 and 2), execute the kernel object (line 4) and finally transfer the result from the device to the host memory (line 6). In order to execute the kernel object, some settings (i.e., *global* and *local* parameters) need to be previously defined. These settings refer to the number and grouping of GPU threads used to execute the functionality.

   Listing 2.6: Sending commands via the *command_queue* object

```
1  clEnqueueWriteBuffer(command_queus, a_in, CL_TRUE, 0, sizeof(
        float) * n, a_data, 0, NULL, NULL);
2  clEnqueueWriteBuffer(command_queue, b_in, CL_TRUE, 0, sizeof(
        float) * n, b_data, 0, NULL, NULL);
3
4  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global, &
        local, 0, NULL, NULL);
5
6  clEnqueueReadBuffer(command_queue, c_out, CL_TRUE, 0, sizeof(
        float) * n, c_res, 0, NULL, NULL );
```

III

Chapter 3

GPU support in component-based systems

In this chapter, we aim to present existing knowledge regarding component-based applications with GPU capability. In the context of the thesis, we introduce this chapter in order to position our work with respect to the existing research. Regarding the research contributions and goals, the chapter presents contribution 1 which addresses RG1.

3.1 Study design

To characterize the state-of-the-art of the usage of CBD for applications with GPU capability, we follow the systematic literature review (SLR) methodology [30][31]. Guidelines for performing SLR in software engineering are introduced by Kitchenham et al. [30], where the following three main phases are suggested, as follows:

1. An SLR starts with the planning phase, which: *i)* describes the motivation to conduct the SLR, *ii)* specifies the research questions to be answered, and *iii)* develops the rules for conducting the SLR.

2. During the second phase, the rules defined in the previous step are followed, primary studies are collected and information is extracted.

3. The reporting of the review is covered in the last phase, where all the information of the SLR, from the motivations to the data synthesis and to the RQs, are presented through a report.

Following these guidelines, we introduce the research process adopted for this study in Figure 3.1, where the three phases have one or several included activities.

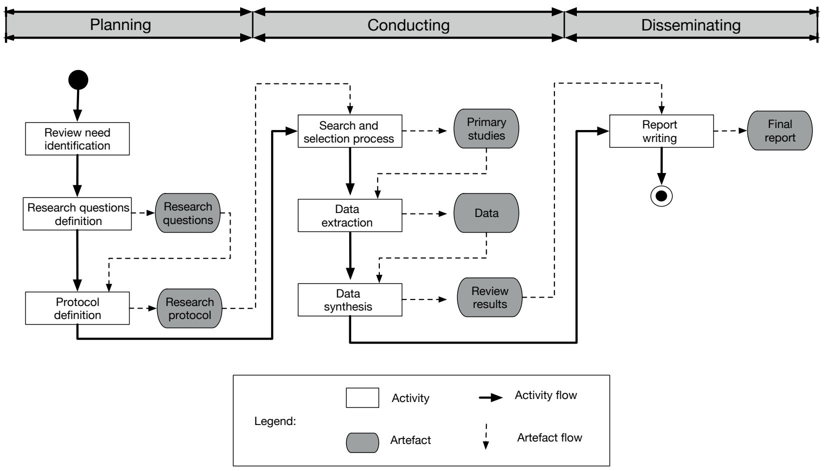


Figure 3.1: Overview of the SLR research process

The following sections describe in depth each phase activity of the followed research process.

3.1.1 Review need identification

Our work aims at introducing GPU support in component-based embedded systems. The goal of this SLR is to identify existing research which targets GPU support in systems that follow a component-based approach, and to summarise this knowledge. The motivation that drives us to conduct this SLR is given by the following reasons:

1. to characterize the state-of-the-art in order to identify and understand the on-going scientific research on GPU support in component-based systems, and
2. in the context of this thesis, we want to examine the work that actually target our problem and to position ourself in the current research.

3.1.2 Research questions definition

We address the goal of the SLR through two research questions, each with a defined objective, as follows:

RQ1 - What are the publication trends of research studies on component-based development of software applications with GPU capability? By providing an answer to this question, our objective is to capture the scientific interest in this subject and its trend, the venues where the results were published and the existing contribution types.

RQ2 - What specific solutions are adopted by existing component-based development approaches when providing GPU support? By answering this question, we aim to present a deeper understanding of the existing research solutions.

3.1.3 Protocol definition

During this activity, we define the steps and rules for conducting the SLR. Basically, we define five steps with their definitions detailed in the following sections. The five steps are the following:

- the resources and search terms used to search for primary studies,
- the selection criteria used to include or exclude studies from the systematic review,

- the data extraction strategy, i.e., how to obtain the required information from each primary study,
- the synthesis of the extracted data, and
- the dissemination of the results.

3.1.4 Search and selection process

The search and selection process has the goal to identify the studies that are relevant to answer the aforementioned research questions. It contains four steps, as follows.

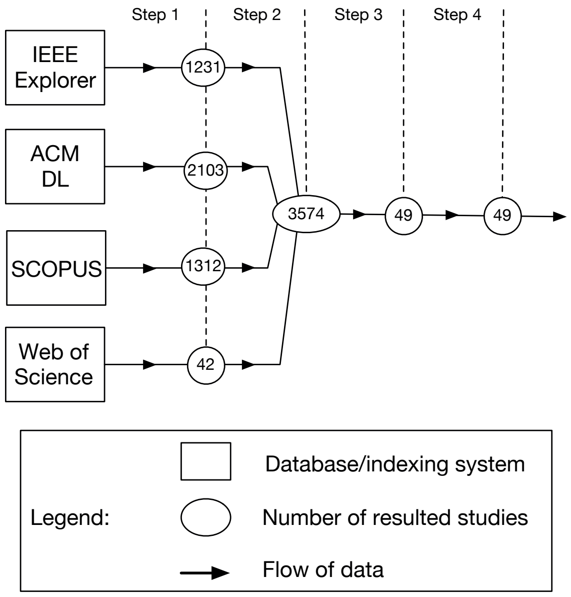


Figure 3.2: Overview of the search and selection process

In the first step, we define: *i)* the databases, and *ii)* the keywords to be used for searching primary studies. In the same step, we start the search process by applying the defined keywords on each database. The second step merges the results found from all databases in a single spread-sheet, and removes the duplicates. During the third step, the merged studies are filtered using a predefined number of (inclusion and exclusion) criteria. Finally, a snowbolling activity is covered by the last step. These steps and their results are summarized in Figure 3.2, where the output of each step is represented by a number of studies.

The four steps of the search and selection process are explained in more details in the following paragraphs.

Step 1. Database search.
We carried out our search on four databases and indexing systems, i.e., IEEE Library, ACM Library, SCOPUS and Web of Science, which are presented in Table 3.1. We considered these sources to be the most relevant ones and suitable for our study due to their high accessibility, their content of many articles in computer science, and their ability to easily export search results to standard formats.

Table 3.1: The databases and indexing systems used in the search process

Name	**Type**	**URL**
IEEE Xplore	Electronic database	http://ieeexplore.ieee.org
ACM DL	Electronic database	http://dl.acm.org
SCOPUS	Indexing system	http://www.scopus.com
Web of Science	Indexing system	http://webofknowledge.com

Considering the two aspects that we want to interplay, i.e., CBD and GPUs, we define a number of keywords and group them in two categories, each category describing an aspect of the review study. The category that targets CBD aspects contains six keywords, while the other category that targets GPU aspects includes three keywords. Table 3.2 presents the nine defined keywords and their corresponding groups. For the last defined keyword (i.e., G9), we use the asterisk symbol in order to capture common used terms such as *CPU+GPU* or *GPU-based*.

Table 3.2: The defined keywords and their corresponding groups

Number	Keyword	Group
C1	component based	CBD aspects
C2	component model	
C3	component oriented	
C4	software component	
C5	cbd	
C6	cbse	
G7	graphical processing	GPU aspects
G8	graphics processing	
G9	*gpu*	

All the search keywords from each group were combined by using the Boolean OR operator, while the two groups were combined using the AND operator. That is, the resulting search string is:
(C1 `OR` C2 `OR` C3 `OR` C4 `OR` C5 `OR` C6) `AND` (G7 `OR` G8 `OR` G9).

For the considered databases, we applied the search string on all the fields of the articles such as the title, author, abstract and full text. In the case of the Web of Science indexing system, there was no possibility to apply the search string on all article fields. Instead, we used the *Topic* field to lead the search, which included the title, abstract and author keywords fields. Furthermore, we noticed that there is a lack of standardization between the selected database sources. Therefore, when searching in all four libraries, we manually adapted the search string for each source to syntactically match their specifications.

The figure introduced in the beginning of this section (i.e., Figure 3.2) presents the four selected libraries and indexing systems. After applying the search string, we obtained a number of 1231 items from IEEE Xplorer, 2103 items from ACM DL, 1312 from SCOPUS and 42 from Web of Science.

Step 2. Merging, and duplicates and impurity removal
The search results from the previous step are combined together into a spreadsheet and the duplicated entries (i.e., with same title, authors, publication venue and year) are removed. Furthermore, we removed entries that were not research papers (i.e., prefaces, forewords, front covers, conference reviews, proceedings, statistic papers, conference presentations, books and book sections, front covers, and table of contents). Given that GPUs were introduced in 1999, we decided to exclude all papers published before the year 2000.

After the merging activity, a number of 3969 entries were available. Summarised by Figure 3.2 in Step 2, the removal of the publications anterior to 2000 led to 3858 entries, and finally the exclusion of non research studies produced 3574 items.

Step 3. Application of the selection criteria
To further filter the primary studies found in the previous step, we define a set of criteria, which are detailed below. A paper will be included if it satisfies all the inclusion criteria, and excluded if it satisfies any of the exclusion criteria.

We define the following six inclusion criteria. A study is included if it:

I1. is full-text available,
I2. describes component-based solutions for developing software,
I3. contains solutions that specifically target platforms with GPUs,
I4. provides an evaluation of the proposed solution,
I5. is peer reviewed, and
I6. is written in English.

There is one exclusion criterion defined as follows:

E1. A study is excluded if it is a secondary or tertiary study (e.g., surveys, systematic literature reviews).

During the beginning of the study, we carried out two trials (i.e., containing 20 and respectively, 50 studies) with the reason to harmonize our understanding of the criteria and/or refine them, if needed. Initially, during the first trial, I2 and I3 were merged into a single criterion, but we decided to separate them into distinct and simpler criteria. Furthermore, in both trials, we discussed the included studies with each study participant, in order to reach a common agreement about the studies to be included.

Applying the inclusion and exclusion criteria on the studies left from step 2, produced 49 research items. We mentioned that the filtering was manually achieved and we used the adaptive reading depth technique [32], that is, in the cases where the title and abstract were not providing enough information to discard or include a study, the full source of the article was browsed through. Furthermore, for studies that were difficult to decide, the study participants decided altogether their inclusion or exclusion from the review study.

Step 4. Snowballing
In this step, we aim to enlarge the set found by the automatic search from previous steps, through a snowballing activity [33]. Through the snowballing activity, we referred to each study and used its reference list to identify additional studies. Several potential studies were identified but after applying the inclusion and exclusion criteria, none of them were introduce in the final set. For example, the study *Component-based design approach for multicore SoCs* by W. Cesrio et al. was excluded from the study because I3 criterion was not fulfilled. This step did not change the number of considered primary studies, i.e., 49 items.

3.1.5 Data extraction

The total number of studies relevant for this work is 49 and they are enumerated in Table 3.13 (i.e., found at the end of this chapter). The papers have an unique id (i.e., from P1 to P49) and are ordered according to their publishing year. The data extraction activity is done manually with the main goal to collect data from the selected primary studies through a *data extraction form*. The form was composed of two aspects, each corresponding to one of the research questions. Specifically, to address RQ1 (i.e., regarding the publication trends), we captured standard information such as the title, authors, publication year and venue. To address RQ2 (e.g., regarding the GPU support of component-based applications), we used a systematic *keywording* approach [32].

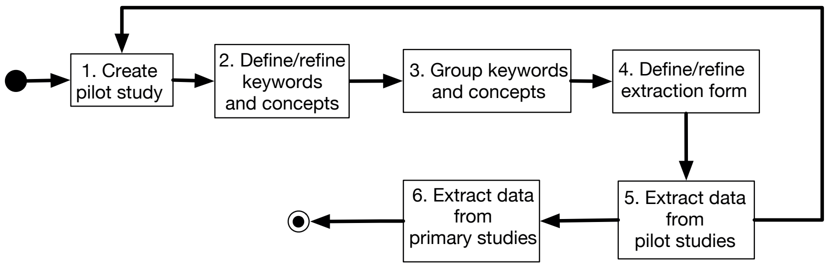

1. Create pilot study
2. Define/refine keywords and concepts
3. Group keywords and concepts
4. Define/refine extraction form
5. Extract data from pilot studies
6. Extract data from primary studies

Figure 3.3: The data extraction process

Figure 3.3 describes in details the process of data extraction used in this study. It contains the following steps:

1. In the first step, we create a test pilot study that contains 20 random studies.
2. In the second step, we define the keywords and concepts representatives for the study[1]. For example, we define *memory* as a keyword, and *the provided memory support* as a concept. If needed, the keywords are refined.
3. The defined keywords and concepts are enclosed in two groups, where a group corresponds to each research questions.
4. In the fourth step, we define the extraction form, or refine it. Basically, this is a table with several columns, where the first two columns represent the id and title of each study. The other columns contain information about e.g., the publication type and year, the domain targeted by the studies, and the type of the used evaluation.
5. Using the defined keywords and concepts, we extracted data from each study of the pilot study and filled in the extraction form. The collected in-

[1]The full list of the defined keywords and concepts can be downloaded using the following link: https://github.com/gcu01/SLR/raw/master/keywords.pdf

formation is reviewed and if we consider that the defined e.g., keywords are not representative enough, we reiterate from step 1.

6. The data from all remaining primary research studies is extracted[2].

3.1.6 Data synthesis

During this activity, the data extracted from the previous step is analyzed and summarized. Using the experience provided by Cruzes et al. [34], we defined two analysis dimensions, i.e., the vertical and horizontal dimensions, as follows:

- The *vertical analysis* presents trends and information about the defined parameters (see Section 3.2 and Section 3.3).

- In the *horizontal analysis*, we explored, through the extracted data, possible relations between the parameters. In this sense, we made comparisons between two or several parameters in order to detect possible relations of various aspects of our study (see Section 3.3).

The narrative synthesis technique [35] is used to explain and interpret the results found from the analysis. The results of this activity are presented in Section 3.2 and Section 3.3.

3.1.7 Threats to validity

The validity threats considered for this work refer to the construction, history record and bias issues. We decrease the first two threats by adopting well-accepted systematic methods to conduct our research process, and documenting all the steps in our research protocol. Furthermore, by providing a well-documented protocol, other researchers that are interested in the research area may replicate our study, or the study design framework can serve as a starting point for new literature review studies.

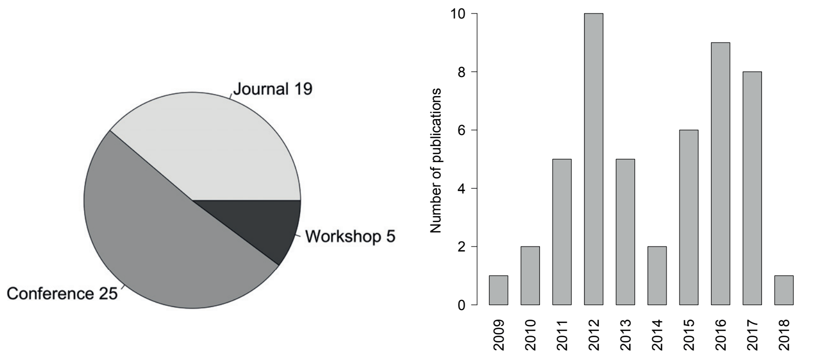
[2]The full extraction form can be downloaded using the following link: https://github.com/gcu01/SLR/raw/master/dataSLR.xlsx

Another dimension that may threaten the validity of our work is the human factor and the bias concern. We decreased it two ways, as follows:

- All members of this study discussed, defined and refined together the keywords, concepts and extraction form.
- We increased the confidence in the inclusion/exclusion criteria filtering by using the Cohen Kappa coefficient [36]. Two trials were executed on two different sample studies, the first one containing 20 random studies, while the second contained 50 random studies. Using the Cohen-Kappa coefficient, we obtained an agreement of 0.3 for the first trial, and 0.6 for the second[3].

3.2 Results - RQ1

This section describes our findings after analyzing the selected primary studies. The findings, illustrated by Figure 3.4 are detailed in the next paragraphs.



(a) The types of publications (b) Number of publications per years

Figure 3.4: The distribution of the publications

[3]The Cohen Kappa statistics show good agreement for anything above 0.6 [37]

Based on their type, we classified the primary studies in three categories, i.e., journals, conferences and workshops. Figure 3.4(a) describes the number of studies for each defined publication category. The most common type of publication is the conference study (i.e., 25/49), followed by the journal publications (i.e., 19/49), and workshop studies (i.e., 5/49).

Figure 3.4(b) presents the publications distribution over the years. We notice that the interest in the development of applications with GPU capability using the component-based approach, started in 2009. The first published paper (i.e., P1) introduces a solution to support specialized visualizations on GPUs, applications that are used in e.g., the medicine domain. From 2010, the interest fluctuates, reaching the peaks in 2012 and 2016.

Up to 2009, our study captured no interest from the research community regarding the GPU support in component-based applications. The growth of the research interest in our studied topic coincides with the release of OpenCL 1.0 (i.e., in 2009) and of several System-on-Chip platforms that combined CPU and GPU onto the same physical chip-set (e.g., Qualcom Adreno[4] released in 2009, NVIDIA Tegra[5] released in 2010 and AMD Fusion[6] released in 2011).

Table 3.3: The research environment

Environment	# Studies	Studies
Academy	48	P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16,P17,P18,P19, P20,P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,P31,P32,P33,P34,P35, P36,P37,P38,P39,P40,P41,P42,P43,P44,P45,P47,P48,P49
Industry	3	P2,P43,P46

Regarding the research environment of the studies, Table 3.3 describes, based on the author affiliation, the studies written in academia and in industry settings. We notice that most of the studies are written in academia, where only three studies are written with industrial context (i.e., P2, P43 and P46).

[4]https://www.pcper.com/reviews/Processors/Qualcomm-History-and-its-GPU-Revolution

[5]http://www.nvidia.com/object/tegra.html

[6]https://www.amd.com/en-us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx

Table 3.4: The domain of the primary studies

Domain	# Studies	Studies
Embedded Systems	17	P5,P6,P11,P19,P20,P22,P24,P32,P33,P35,P36,P40,P42, P43,P44,P45,P47
General purpose	16	P4,P6,P8,P9,P10,P12,P16,P21,P23,P25,P27,P29,P30,P37, P48,P49
High-Performance Computing	5	P2,P6,P7,P38,P47
Computer graphics	4	P1,P17,P18,P41
Physics simulations	4	P14,P28,P31,P39
Medicine simulations	2	P3,P26
Mathematics applications	1	P15
Military applications	1	P46
Pattern matching	1	P34
Video encoding	1	P13

Furthermore, our study captured 10 domains that adopt a component-based approach for applications with GPU capability. Table 3.4 presents the specific studies that target the 10 domains. The top most targeted domains are the embedded systems with 17 studies, and general purpose with 16 studies. We mention that a study may target one or several domains. For example, P6 targets three domains, i.e., embedded systems, general purpose and high-performance computing.

For the evaluation part, the primary studies are divided into three categories, each one using a different type of evidence. The three distinguished types of evidence are:

- *example* which refers to one in-house (not industrial) example;
- *set of examples* referring to several in-house examples; and
- *industrial example* that refers to an example that comes from industry and performed in laboratory.

As presented by Table 3.5, the most used category is the example-based evaluation, where each of the 25 studies applies its proposed solution on a in-house example. The second most used category which is based on a set of examples, is utilized by 22 studies, while the third category used by two primary studies, apply their solution on an industrial example. The extracted data related to the evaluation part, shows that the research on CBD and GPU is rather young, using only three evidence types.

Table 3.5: The type of evaluation used by the primary studies

Evaluation	# Studies	Studies
Example	25	P3,P4,P6,P7,P10,P14,P19,P20,P22,P23,P24,P28,P29,P30,P31, P32,P34,P36,P38,P39,P40,P41,P42,P44,P45
Set of examples	22	P1,P2,P8,P9,P11,P12,P13,P15,P16,P17,P18,P21,P25,P26,P27, P33,P35,P37,P46,P47,P48,P49
Industrial example	2	P5,P43

Table 3.6 presents the venues of the research studies. The research regarding component-based applications with GPU capability is spread over 44 different venues, where the table describes the venues that have more than one study.

Table 3.6: The publication venues

Publication venue	# Studies
The Journal of Parallel Computing	3
The Int. Conf. of High Performance Computing, Networking, Storage and Analysis (SCC)	2
The Journal of Procedia Computer Science (ICCS)	2
The EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)	2
The Journal of Concurrency and Computation	2
Others	38

Based on the presented extracted data, we can conclude that the research interest on GPU and CBD is rather recent, scattered and immature, and it is not a mainstream practice in industry.

The highlights of RQ1 are the following:

- ▶ Up to 2009, there is no interest in the component-based applications with GPU capability. From 2009, there is an increased interest, which may have been triggered by the release on the market of several (software and hardware) technologies.
- ▶ Most of the studies are researched in academic settings, covering a large variety of domains.
- ▶ Research on component-based applications with GPU capability is spread over a large number of heterogeneous venues (i.e., more than 40) with a focus on conference and journal type of publications.

3.3 Results - RQ2

This section presents our findings regarding RQ2. From each study, we have extracted information to cover various research facets which are presented in the form of tables and figures, in the following paragraphs.

The first research aspect that we introduce is related to the challenges targeted by the primary studies. We distinguish the following four categories of challenges:

- *Development improvement* refers to studies that aim to facilitate the development of applications.
- *Performance improvement* indicate the studies that focus on improving the application performance.
- *Allocation* represents studies that handles the software-to-hardware allocation.
- *Experience sharing* presents studies that share their obtained knowledge.

Table 3.7 presents the main challenges targeted by the primary studies. The most addressed challenge is development improvement (i.e., 33 studies). For

Table 3.7: The main challenges targeted by the primary studies

Problem	# Studies	Studies
Development improvement	33	P1,P2,P4,P6,P7,P8,P9,P10,P11,P13,P14,P16,P17,P18,P20, P21,P22,P26,P27,P28,P29,P30,P31,P33,P34,P35,P36,P38, P39,P40,P42,P47,P49
Performance improvement	10	P3,P5,P12,P15,P19,P25,P37,P41,P46,P48
Allocation	5	P23,P24,P32,P44,P45
Experience sharing	1	P43

example, P7 facilitates the development of high-performance computing applications through software components that can be reused across different architectures and resource configurations. The performance improvement (i.e., 10 studies) is the second most addressed challenge. In this category, P3, for example, targets the performance of disease simulations, delivering a 4 times speedup when using a component-based approach. The allocation challenge, addressed by five studies, refers to studies that cover the software-to-hardware distribution. There is one study (i.e., P43) that presents the experience gathered in industry, where the GPU is used to reduce the worse-case execution time in the powertrain domain.

Table 3.8: The utilized component-based approach

Component models	# Studies	Studies
No specific CM	34	P1,P2,P7,P8,P12,P13,P14,P15,P16,P17,P18,P22,P23,P24,P25, P26,P27,P28,P29,P30,P31,P32,P33,P34,P38,P39,P41,P43,P44, P45,P46,P47,P48,P49
PEPPHER	6	P6,P9,P10,P11,P21,P37
UML	4	P5,P19,P20,P40
CCA	2	P3,P4
Rubus	2	P35,P42
OSGi	1	P36

Regarding the component-based approach, we divided the research studies into two categories, one that uses a general approach and the other that uses specific component models, as described in Table 3.8. While there are 34 studies in the former category, 15 studies adopt specific component models. The most utilized component models are PEPPHER [38], UML[7], CCA [39] and Rubus [6]. Although UML is not a component model per se, we included the studies with UML-based solutions in the category with specific component models. The inclusion reason is due to the fact that studies construct their component-based solution using the UML description. Using the same reasoning, we also included in this category all the studies that use UML extensions. For instance, P19 uses the MARTE extension to develop embedded systems with real-time characteristics.

Table 3.9: The used approaches in providing GPU support

Mechanism	# Studies	Studies
Programming	43	P1,P2,P3,P4,P6,P8,P9,P10,P11,P12,P13,P14,P16,P17,P19, P20,P21,P22,P23,P24,P25,P26,P27,P28,P30,P31,P32,P33, P34,P35,P36,P37,P38,P39,P40,P41,P42,P43,P44,P45,P46, P47,P48
Modeling	20	P1,P2,P3,P4,P5,P14,P18,P19,P20,P23,P24,P29,P32,P33,P35, P36,P40,P42,P44,P45
Library	13	P6,P10,P11,P12,P18,P21,P25,P30,P33,P39,P41,P42,P47
Template	11	P7,P10,P11,P12,P15,P22,P34,P38,P39,P42,P47
Annotations	10	P6,P8,P9,P10,P11,P17,P21,P25,P47,P49
Model-to-code generation	8	P5,P14,P18,P19,P20,P33,P35,P40
Code-to-code generation	7	P8,P9,P10,P11,P15,P17,P21
Model-to-model generation	2	P5,P40
Grammar	1	P49
Parametrization	1	P29
Layer	1	P7

Table 3.9 presents the approaches used to address the GPU support. Programming is the most used approach, followed by modeling and library-usage. A study may use a combination of different approaches, such as P10 that combines programming, library- and template-usage, annotations and code-to-code

[7]http://www.uml.org

generation. We notice that model-to-model and model-to-code generations are always use as extensions of the modeling approach.

Table 3.10: The memory mechanisms adopted by the primary studies

Memory mechanism	**# Studies**	**Studies**
Specific artifact	18	P5,P6,P7,P10,P11,P12,P15,P17,P18,P22,P33,P34,P35,P38, P39,P42,P47,P48
Not available	15	P3,P13,P21,P23,P24,P26,P29,P32,P36,P37,P41,P43,P44,P45, P49
Manual	9	P1,P14,P25,P27,P28,P30,P31,P34,P46
Layer	6	P2,P4,P9,P16,P19,P30
Model transformation	3	P8,P20,P40

The memory aspect is an important issue when utilizing GPUs. We identified the following four specific categories that deal with it:

- *Specific artifact* refers to studies that use items specifically developed to address the system memory.
- *Manual* indicate the studies that manually address the memory of the system.
- *Layer* represents studies that address the system memory via a layer.
- *Model transformation* presents studies that use model transformation mechanisms.

Table 3.10 presents the specific memory mechanisms adopted by the primary studies to address the GPU memory aspect. There are 34 primary studies that specifically address this aspect, and 15 studies that do not mention the memory support in their solution. The most popular solution (i.e., 18 studies) is to use a specific artifact. For example, P18 introduces specific blocks (i.e., classes) to handle various data activities such as copy and resizing. Other approaches include manual implementation (i.e., 9 studies), layer-based approach (i.e., 6 studies) and using model transformation mechanisms (i.e., 3 studies). We notice that there are two studies that use combination of approaches, i.e., P30 that combines the manual and layer-based approach, and P34 that uses a specific artifact with manual implementation.

Table 3.11: The programming languages used by the primary studies

Programming language	# Studies	Studies
C/Cpp	36	P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P14,P15,P16,P17, P18,P19,P20,P21,P22,P25,P26,P27,P28,P30,P31,P32,P33, P34,P35,P39,P40,P41,P42,P47
CUDA/OpenCL	34	P2,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P17,P18,P19,P20, P21,P22,P25,P26,P27,P28,P30,P31,P32,P34,P35,P36,P40, P42,P43,P46,P47,P48,P49
OpenMP	12	P3,P4,P5,P6,P10,P15,P17,P19,P21,P22,P31,P33
MPI	6	P4,P15,P16,P31,P38,P39
OpenGL	3	P1,P26,P41
Java	2	P4,P36
Fortran	2	P4,P5
OpenACC	2	P28,P31
Vulkan	1	P41
Csharp	1	P38
VHDL	1	P5
OpenVX	1	P33

The programming languages used by the primary studies are presented in Table 3.11. The two most used (group of) languages are C/Cpp (i.e., 36 studies) and CUDA/OpenCL (i.e., 34 studies). We want to mention that a study may use one or several languages such as P6 that allows its solution to use C/Cpp, CUDA/OpenCL and OpenMP. Another fact to notice is that most of the studies that utilize CUDA/OpenCL also use C/Cpp. This is normal because CUDA/OpenCL are built using the C/Cpp syntax which allows them to be easily used together.

Concerning the GPU platform, we distinguish between studies that target multi-GPU and single-GPU platforms, as described in Table 3.12. The

Table 3.12: The type of platform targeted by the primary studies

Type platform	# Studies	Studies
Multi-GPUs	25	P2,P5,P8,P10,P14,P18,P23,P24,P28,P29,P30,P31,P33,P34, P37,P38,P39,P40,P42,P43,P45,P46,P47,P48,P49
Single GPU	21	P6,P7,P9,P4,P3,P11,P13,P15,P16,P17,P19,P20,P22,P25, P26,P27,P32,P35,P36,P41,P44

multi-GPUs platforms refer to systems that contain two or more GPUs, and the single-GPU platforms represent systems with single GPUs. While the former category contains 25 studies, the later one comprises 21 studies. We mention that three studies did not specify the type of GPU platform targeted.

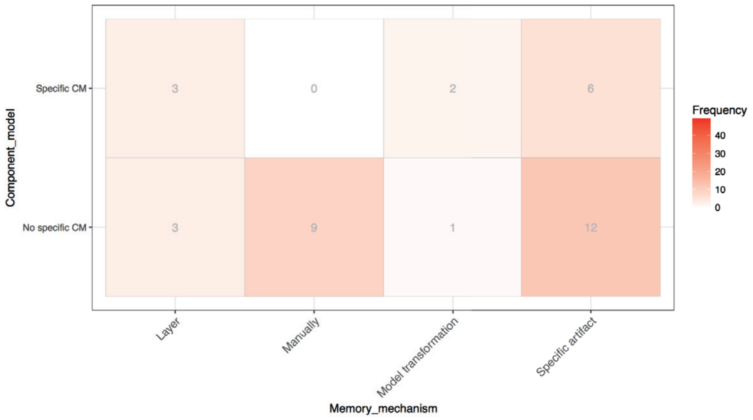


Figure 3.5: The memory mechanisms used by the primary studies

Figure 3.5 presents the type of memory mechanisms used by the primary studies. We notice that, for the solutions that do not use specific component models, the most utilized mechanisms are the specific artifacts and the manual manner to address them. In the case of the solutions that use specific component models, there are no study that manually addresses the memory aspect. In these cases, the most utilized method is through specific artifacts. A study may combine several mechanisms to address the GPU memory. For example, P34, based on a non specific component-model approach, uses a combination of specific artifacts and manual development.

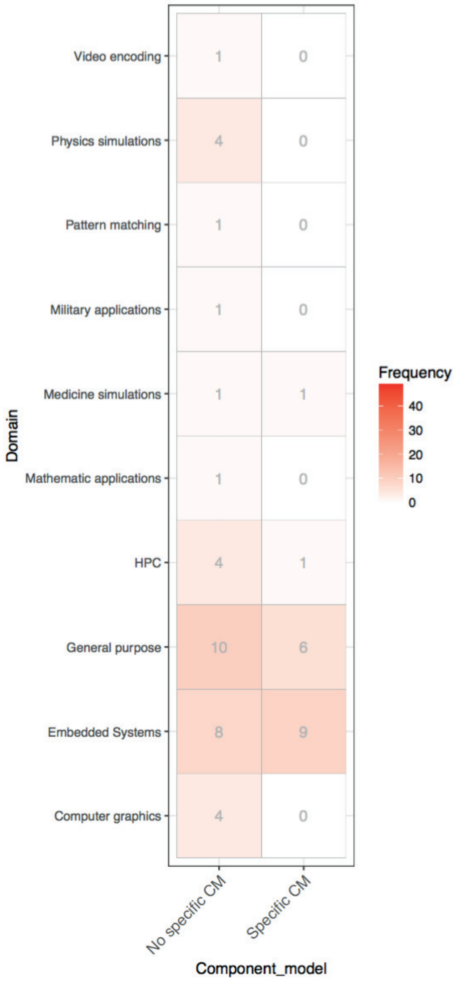


Figure 3.6: The domains targeted by the primary studies

There are 10 domains targeted by the primary studies, which are presented by Figure 3.6. Nine of the domains are specific (e.g., medicine simulations) while one of the domain is generic given that studies do not target particular domains. The studies that use no specific component models cover all the reported domains, with a focus on embedded systems, computer graphics and physics simulations. The studies with specific component models target three specific domains, with a focus on embedded systems. A study may target several domains, such as P6 which focuses on embedded systems and high-performance computing.

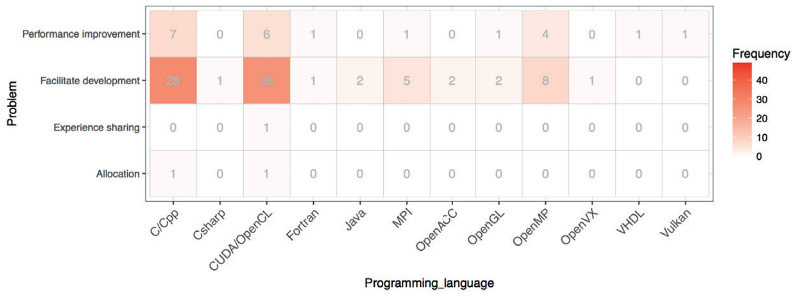


Figure 3.7: The programming languages and the addressed challenges

Figure 3.6 illustrates the programming languages used to address the challenges targeted by the primary studies. It is not a surprise that the GPU specialized languages (i.e., CUDA/OpenCL) are most used, alongside with C/Cpp. Given that CUDA/OpenCL are built using the C/Cpp syntax, these programming languages are frequently used together, hence most of the studies that utilize CUDA/OpenCL, also use C/Cpp. Other used languages are MPI and OpenMP, which are also used in combination with C/Cpp due to their construction (i.e., using C/Cpp syntax).

The last figure (i.e., Figure 3.8) illustrates the mechanisms used to support GPU solutions, and the types of platforms (i.e., with one or multi-GPUs). The utilized mechanisms are evenly used for the two types of platforms. Programming is the most utilized mechanism to develop solutions with GPU capability, either on platforms with a single or multiples GPUs. Other utilized mechanisms include modeling, libraries and templates.

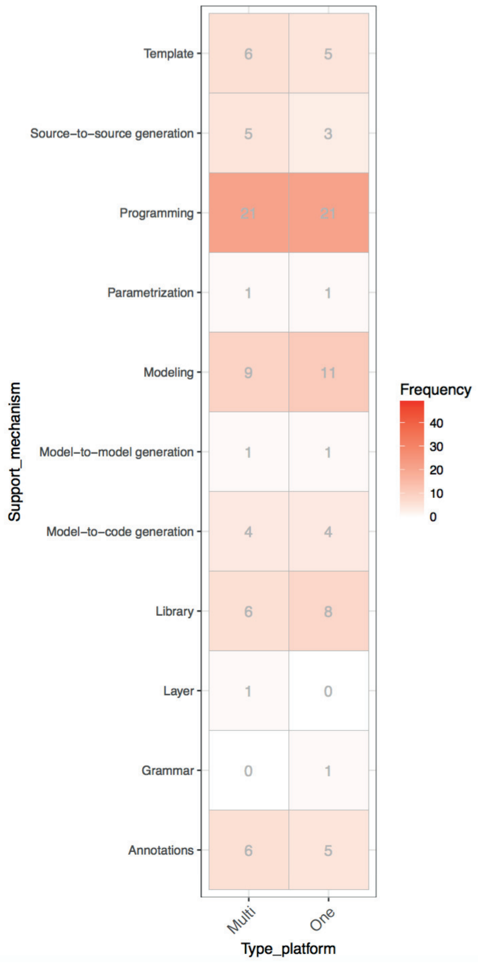


Figure 3.8: The GPU support mechanisms and the types of platforms

The highlights of RQ2 are the following:

- ▶ The captured studies focus on development and performance improvement.
- ▶ Most of the studies do not use specific component models to construct their solutions. For the studies that use specific component models, PEPPHER, UML, CCA and Rubus are the most utilized.
- ▶ More than 10 mechanisms are used to support GPU development. Programming and modeling mechanisms are the most used ones.
- ▶ A specific artifact is the most used mechanism to address the memory issue. Other mechanisms include manual development and layer-based solutions.
- ▶ The most used programming languages are CUDA/OpenCL and C/Cpp. Often, CUDA/OpenCL and C/Cpp are used together.

Table 3.13: The list with the primary studies

Id	Title	Author	Year
P1	Voreen: a rapid-prototyping environment for ray-casting-based volume visualizations	J. Meyer-Spradow et al.	2009
P2	2PARMA: parallel paradigms and run-time management techniques for many-core architectures	C. Silvano et al.	2010
P3	Component-based design for adaptive large-scale infectious disease simulation	T.M. Riechersm et al.	2010
P4	A model-driven design framework for massively parallel embedded systems	A. Gamatie et al.	2011
P5	Improving programmability of heterogeneous many-core systems via explicit platform descriptions	M. Sandrieser et al.	2011
P6	PEPPHER: efficient and productive usage of hybrid computing systems	S. Benkner et al.	2011
P7	Reusable software components for accelerator-based clusters	M.M. Rafique et al.	2011
P8	Scheduling multi-paradigm and multi-grain parallel components on heterogeneous platforms	Y. Pen et al.	2011
P9	Computing effective properties of random heterogeneous materials on heterogeneous parallel processors	T. Leidi et al.	2012
P10	High-efficient parallel CAVLC encoders on heterogeneous multicore architectures	H Su et al.	2012
P11	High-level specifications for automatically generating parallel code	A. Acosta et al.	2012
P12	High-level support for pipeline parallelism on many-core architectures	S. Benkner et al.	2012
P13	Improving performance of adaptive component-based dataflow middleware	T.D.R. Hartley et al.	2012
P14	Leveraging PEPPHER technology for performance portable supercomputing	C. Kessler et al.	2012
P15	Policy-based tuning for performance portability and library co-optimization	D. Merrill et al.	2012
P16	Rapid prototyping of image processing workflows on massively parallel architectures	B. Li et al.	2012
P17	The PEPPHER composition tool: performance-aware dynamic composition of applications for GPU-based systems	U. Dastgeer et al.	2012
P18	Using explicit platform descriptions to support programming of heterogeneous many-core systems	M. Sandrieser et al.	2012

P19	A framework for performance-aware composition of applications for GPU-based systems	U. Dastgeer et al.	2013
P20	An extended model for multi-criteria software component allocation on a heterogeneous embedded platform	I. Svogor et al.	2013
P21	Programmability and performance portability aspects of heterogeneous multi-/manycore systems	C. Kessler et al.	2013
P22	System synthesis from UML/MARTE models: The PHARAON approach	H. Posadas et al.	2013
P23	Towards code generation from design models for embedded systems on heterogeneous CPU-GPU platforms	F. Ciccozzi	2013
P24	Component allocation optimization for heterogeneous CPU-GPU embedded systems	G. Campeanu et al.	2014
P25	Performance portability across heterogeneous SoCs using a generalized library-based approach	S. Fang et al.	2014
P26	A software component approach for GPU accelerated physics-based blood flow simulation	J. Wu et al.	2015
P27	Lattice QCD code bridge++ on arithmetic accelerators	S. Motoki et al.	2015
P28	Modeling and analysis of performances for concurrent multithread applications on multicore and graphics processing unit systems	D. Cerotti et al.	2015
P29	OpenCL vs OpenACC: Lessons from development of lattice QCD simulation code	H. Matsufuru et al.	2015
P30	Parallel programming in actor-based applications via OpenCL	P. Harvey et al.	2015
P31	Towards a high performance geometry library for particle-detector simulations	J. Apostolakis et al.	2015
P32	A 2-layer component-based architecture for heterogeneous CPU-GPU embedded systems	G. Campeanu et al.	2016
P33	Automatic synthesis of heterogeneous CPU-GPU embedded applications from a UML profile	F. Ciccozzi	2016
P34	Contextual abstraction in a type system for component-based high performance computing platforms	F.H. de Carvalho Junior et al.	2016
P35	Domain-specific programming technology for heterogeneous manycore platforms	A. Syschikov et al.	2016
P36	Extending the Rubus component model with GPU-aware components	G. Campeanu et al.	2016
P37	Generic approach for pattern matching with OpenCL	T. Fekete et al.	2016

P38	Hardware-Based efficiency advances in the EXA-DUNE project	P. Bastian et al.	2016
P39	Middleware platform for distributed applications incorporating robots, sensors and the cloud	E. De Coninck et al.	2016
P40	Prunning strategies in adaptive off-line tunning for optimized composition of components on heterogeneous systems	L. Li et al.	2016
P41	Developing CPU-GPU embedded systems using platform-agnostic components	G. Campeanu et al.	2017
P42	Massive parallelization of real-world automotive real-time software by GPGPU	C. Hartmann et al.	2017
P43	Parallel execution optimization of GPU-aware components in embedded systems	G. Campeanu	2017
P44	Run-time component allocation in CPU-GPU embedded systems	G. Campeanu et al.	2017
P45	Shader components: modular and high performance shader development	Y. He et al.	2017
P46	Using high level GPU tasks to explore memory and communications options on heterogeneous platforms	C. Liu et al.	2017
P47	Using OpenCL to increase SCA application portability	S. Bernier et al.	2017
P48	VectorPU: A generic and efficient data-container and component model for transparent data transfer on GPU-based heterogeneous systems	L. Li et al.	2017
P49	Formalised composition and interaction for heterogeneous structured parallelism	M. Goli et al.	2018

IV

Chapter 4

GPU-aware mechanisms

The component-based development methodology is successfully adopted by industry in the construction of embedded systems. Due to the fact that the existing component models used in the embedded system domain offer no GPU support, many shortcomings arise when CBD is used to construct embedded systems with GPUs. Because of these shortcomings, CBD may become unattractive in the context of embedded systems with GPUs. The role of this chapter is:

- to introduce the existing challenges when developing component-based embedded systems with GPUs, and

- to present our solutions to facilitate the development of embedded systems with GPUs.

The solutions, initially introduced as theoretical concepts, are implemented in an existing component model, describing their feasibility aspects.

Regarding the research goals and contributions presented in the Introduction, this chapter addresses the Contribution 2 and Contribution 3 that both target Research Goal 2.

4.1 Running case

In order to describe the detailed challenges of our work, and to introduce and evaluate our solutions, we use an existing underwater robot as a running case. The robot autonomously navigates under water, and, based of its vision system, fulfills various missions, such as tracking and touching red buoys. Similar to most of the existing embedded systems, the robot contains sensors, actuators and an embedded board. As sensors, we mention a system of cameras that provides a continuous flow of data regarding the underwater environment. The thrusters of the robot are its actuators and allow it to move underwater. The embedded board contains a CPU and GPU that are integrated on the same physical chip.

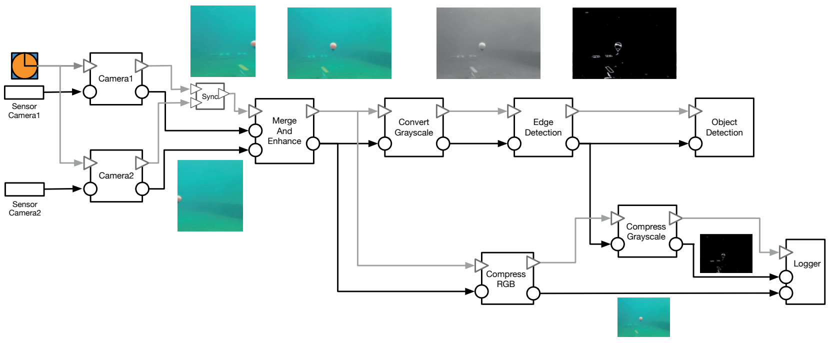


Figure 4.1: The component-based vision system

Figure 4.1 describes the vision system of the underwater robot; it is constructed using the Rubus component model. The system contains nine components, as follows. The *Camera1* and *Camera2* components receive raw data from the camera sensors and convert them to readable frames. The resulting frames are merged by the *MergeAndEnhance* component. The component also removes the noise of the merged frame, and forwards its result to the *ConvertGrayscale* component that filters it into grayscale format. The *EdgeDetection* component receives as input the grayscale frame and provides a black-and-white frame, where the white lines delimits the objects found in the frame.

Based on the delimited objects, the *DetectObject* component takes appropriate actions.

Another part of the vision system is comprised of three components, illustrated in the lower part of Figure 4.1. Two of the components (i.e., *Compress-RGB* and *CompressGrayscale*) compress the frames of the system, while the *Logger* component, that has the purpose to record the underwater journey of the robot, registers the compressed frames.

The vision system contains five components that may be executed on the GPU, due to their functionality, i.e., image processing. These components are *MergeAndEnhance, ConvertGrayscale, EdgeDetection, CompressRGB* and *CompressGrayscale*. By executing them on the GPU, the system may improve one or several aspects such as overall performance.

4.2 Existing challenges

The existing component models used to construct embedded systems provide no specific support for developing applications with GPU capability. Using the existing approaches, one way to construct component-based embedded systems with GPUs is to encapsulate, inside the component, all the GPU-specific information. More specifically, each component needs to encapsulate all the specific GPU information and operations in order to address the hardware platform. Whenever a component uses the GPU to execute its functionality, it needs to e.g., shift its required input data from the main memory system (i.e., the CPU address space) onto the GPU memory system, (i.e., GPU address space). After finishing its processing activities, the result also needs to be shifted back onto the main memory system. Therefore, specific transfer operations and memory initialization mechanisms need to be enclosed in the component.

Encapsulating GPU information inside components brings the following disadvantages:

- inefficient component communication,

- a reduced component reusability,
- a decreased component maintainability,
- a reduced design flexibility, and
- a complex component development process, error-prone and time-consuming.

To present more details on the existing challenges of component-based development of embedded systems with GPUs, we use the underwater robot running case developed using the Rubus component. Figure 4.2 describes the vision system executed on a platform with GPU. The platform has distinct memory addresses for the CPU and GPU.

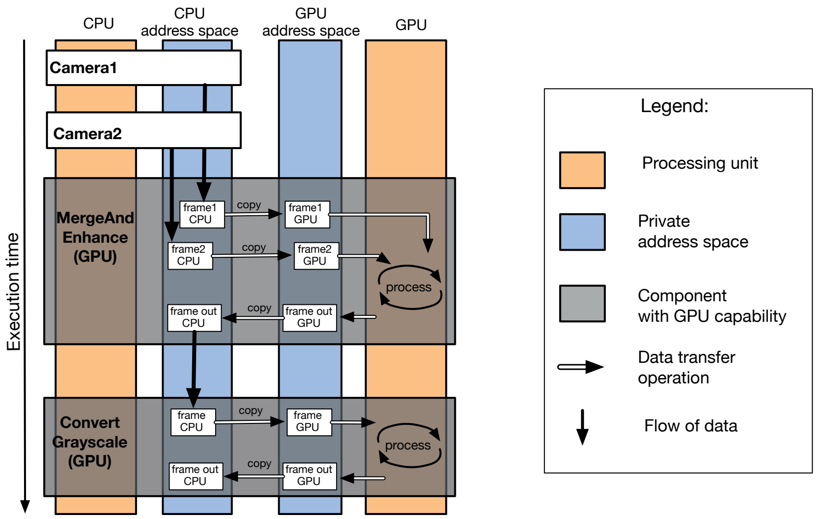


Figure 4.2: Platform-related activities of the vision system

In its current state of specifications, the Rubus component model needs to follow the same approach when addressing GPU-based hardware. Each component with GPU capability needs to encapsulate all the GPU-specific information and operations, such as data transfer operations or the GPU computation settings. Both *Camera1* and *Camera2* components access the main RAM system to acquire the raw image frames, process them on the CPU and store them back onto the RAM. Having GPU computation, the hardware activities of the *MergeAndEnhance* component are different. In addition to the RAM access to obtain the cameras' frames, the component needs to copy them onto the GPU memory system in order to process them. Once the frames are merged and processed, the component copies the result back onto the RAM system.

For a platform in which the CPU and GPU share the same memory, the transfer activities are not required to be encapsulated inside e.g., the *MergeAndEnhance* component. This reduces the reusability of components with GPU capability between contexts. For example, the *MergeAndEnhance* component that is constructed to be executed on a share-memory type of platform, cannot be used on a platform with distinct CPU and GPU memory spaces.

Moreover, the component developer needs to encapsulate inside the components: *i)* environment information to access the GPU, and *ii)* settings regarding the GPU threads usage. These settings which basically refer to the number of GPU threads required by the component, are influenced by the physical limitations of the platform resources. Hard-coding these settings inside the component leads to a decreased maintainability of the component. Furthermore, it influences also the reausability aspect, the component being only reusable on platforms that posses enough resources to execute it.

The component developer, when constructing the component with GPU capability, needs to address, besides the component functionality, all the required environment information and GPU settings. This fact leads to a complex component development process, error-prone and time-consuming.

Another shortcoming that results from the existing way of developing component-based embedded applications with GPU capability, is the fact that the CBD separation of concerns principle between component and system development is broken. When the component developer constructs the component with GPU capability, the encapsulated settings are sett up by inspecting the platform characteristics and software architecture (i.e., which components access the GPU).

4.3 The development process overview

The novelties of this work are introduced using our overview of the development process, as depicted in Figure 4.3. Basically, the development process is composed of: *i)* the component development activity, *ii)* the system development activity, *iii)* the optimization realization activity, and *iv)* the source-code generation activity. In the component development activity, the component developer constructs the components, which are used during the system development activity to build the application architecture. During the optimization realization activity, the application is optimized and realized into a Rubus-like application (i.e., contains only Rubus constructs), while in the last activity, the application is transformed into source-code. In the following paragraphs, the development process is described in more details.

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

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

After constructing the components, the focus of the development process is shifted on the system development activity. The design of the application is executed during the software design step, by composing the already constructed (regular and flexible) components. The output of the software design step is a component-based architecture of the desired software application. Further-

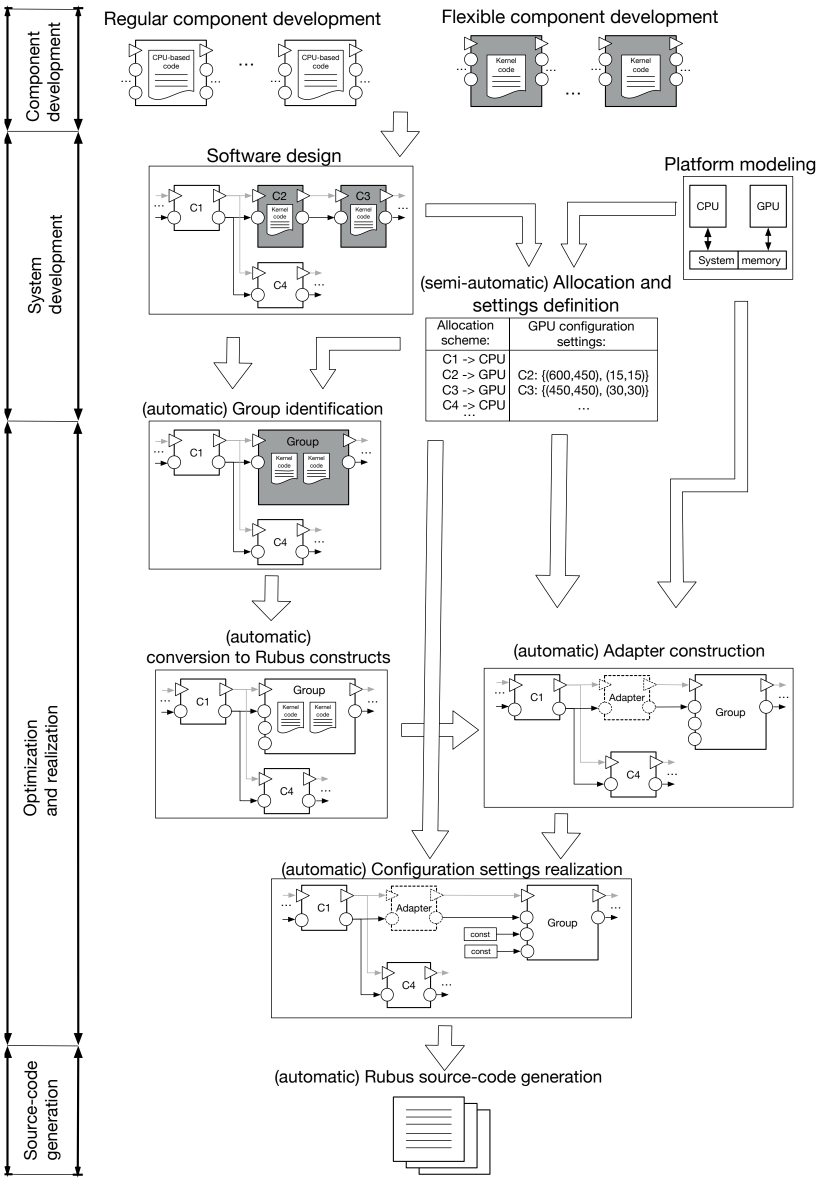

Component development
System development
Optimization and realization
Source-code generation
Regular component development
CPU-based code
Flexible component development
Kernel code
Software design
C1
C2
C3
C4
Platform modeling
CPU
GPU
System
memory
(semi-automatic) Allocation and settings definition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

Figure 4.3: The development process using our solutions

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

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

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

4.4 Flexible component

We consider in this work that a system is design by composing regular and flexible components. While a regular component is always executed on the CPU, a flexible component may be executed by either the CPU or GPU. Formally, we define a system being a set of regular and flexible components, and their connections:

$$\begin{aligned} S &= \langle C_{\mathrm{R}}, C_{\mathrm{F}}, \mathit{Connect} \rangle, \text{ where} \\ C_{\mathrm{R}} &= \{C_{\mathrm{reg}_1}, C_{\mathrm{reg}_2}, ..., \}, \\ C_{\mathrm{F}} &= \{C_{\mathrm{flex}_1}, C_{\mathrm{flex}_2}, ..., \}, \\ \mathit{Connect} &= \{\langle p_{\mathrm{from}_1}, p_{\mathrm{to}_1} \rangle, \langle p_{\mathrm{from}_2}, p_{\mathrm{to}_2} \rangle, ...\}. \end{aligned}$$

We define a component, either regular or flexible, as a tuple that contains the functionality F disclosed through the interface I. All of the components use port-base interaction style and the underlying component model uses a pipe-and-filter style.

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

F denotes the functionality. There is a difference between the functionalities of regular and flexible components (described later on).

I defines a set that contains all (data and trigger) ports.

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

$$\forall p_{\mathsf{k}} \in I,\ p_{\mathsf{k}}.name \text{ denotes the name of port } p_{\mathsf{k}}, \text{ and}$$
$$p_{\mathsf{k}}.type \text{ denotes the type of port } p_{\mathsf{k}}.$$

The ports of a component support data of regular type such as *integer* or *double*. Besides regular data, we introduce *m-elem* as a data type, referred as multi-element type, to describe large data that is also supported by the 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.

Furthermore, we define additional port information related to the multi-element ports to facilitate the generation to source-code activity. The data of the multi-element type may have several dimensions. Therefore we introduce additional attributes to accurate describe multi-element data, as follows:

$$\forall p_{\mathsf{k}}, \text{ where } p_{\mathsf{k}}.type = \textit{m-elem}$$
$$p_{\mathsf{k}}.width \text{ denotes the horizontal size,}$$
$$p_{\mathsf{k}}.height \text{ denotes the vertical size, and}$$
$$p_{\mathsf{k}}.size \text{ denotes the size of each data element.}$$

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), $p_{\mathsf{k}}.height$ 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.

Moreover, another information required during the source-code generation activity is related to the connections of a port. Therefore, we introduce a way to access this information, as follows:

$$\forall p_{\mathsf{k}}, \text{ where } p_{\mathsf{k}}.type = \textit{m-elem} \wedge p_{\mathsf{k}}.type \neq \textit{m-elem}$$
$$p_{\mathsf{k}}.connect \text{ denotes the port(s) connected to } p_{\mathsf{k}}.$$

For an output port that is connected to one or several other ports, the *connect* construct specifies the connected ports. Similarly, for an input port, the construct provides the (only) port connected to it.

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). We mention also that a (regular or flexible) component contains one single input trigger port and one single output trigger port. Therefore:

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

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

Due to the characteristics of the GPU, large data (e.g., images) need to reside on the GPU memory (e.g., for platforms with distinct memory systems) in order to be processed by the GPU. The motivation of introducing the multi-element type and the separation of the interface into two subsets is that automatically generated mechanisms (i.e., adapters) 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 a component is its functionality F. In the case of a flexible component, the functionality is basically the kernel code constructed using the

Listing 4.1: Declaration of the *m-elem* type

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

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 4.1 outlines the construction of the multi-element type.

Concerning the actual implementation of the functionality of a flexible component, 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, 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

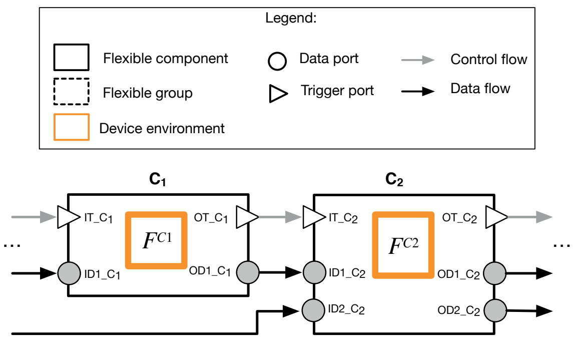
`p_k.name_data` to access the data memory location.

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

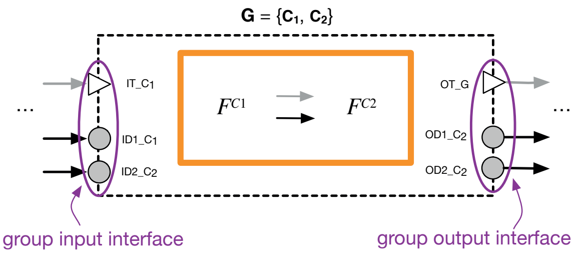
`p_k.name_data` to access the corresponding (one value) buffer.

4.5 Optimized groups of flexible components

In order to reduce the overheads introduced by the flexible component notion, we propose to group connected flexible components with the same hardware allocation, into what we call a flexible group. The flexible group is realized as a single component that has the same allocation as the enclosed components.



(a) The realization of two flexible components



(b) The realization of a flexible group enclosing two components

Figure 4.4: The transition from flexible components to flexible group

Our solution is described through a simple example illustrated in Figure 4.4. Assuming that we have two connected flexible components, C_1 and C_2, and is decided for both of them to be allocated on the GPU. During the source-code generation activity, C_1 and C_2 are realized as regular components; the resulted source-code is graphically described in Figure 4.4a. Each component is realized to encapsulate its own device-environment through which it executes the functionality. The communication between the components functionalities is done through the components port-based interfaces.

With our approach, we optimize the realization of the two components via a flexible group *G*, as depicted by Figure 4.4b. In our vision, *G* contains one OpenCL device-environment through which, the functionalities of the enclosed components are executed. Furthermore, the communication (i.e., data and control flow connections) between C_1 and C_2 is done at the functionality level, i.e., directly between F^{C1} and F^{C2}, by providing the output of F^{C1} as input of F^{C2}. The encapsulation of F^{C1} and F^{C2} inside *G* does not break the (functionality) encapsulation of C_1 and C_2.

The group inherits the input and output ports of the enclosed components, that communicate with anything outside the group. Therefore, the group input interface contains two input data ports, i.e., $ID1\_C_1$ from C_1 and $ID2\_C_2$ from C_2, since these two ports receive data from outside the group. The group output interface has $OD1\_C_2$ and $OD2\_C_2$ data ports, both inherited from C_2. Furthermore, the group is equipped with one input and one output trigger ports.

The remaining of the section presents the definition of the flexible group and the algorithm that identifies groups in a given component-based system.

4.5.1 Definition

We define a flexible group G as an ordered set of connected flexible components that have the same hardware allocation. The order of the components enclosed in the set, determines their execution pattern:

$$G = \langle C_1, C_2, ...\rangle, \text{ where,}$$
$$\text{alloc}(G) = \text{alloc}(C_1) = \text{alloc}(C_2) = ..., \text{ and}$$
$$C_{\mathsf{i}} \text{ executes before } C_{\mathsf{j}} \text{ for } \forall i < j.$$

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

$$I^{\mathsf{G}} = \{p_1, p_2, ...\}, \text{ where } p_{\mathsf{m}} \in I^{\mathsf{C}_1} \cup I^{\mathsf{C}_2} \cup ... \cup \{p^{\mathsf{G}}_{\text{trigg\_in}}, p^{\mathsf{G}}_{\text{trigg\_out}}\}$$

The group interface is divided in two elements, one for the input ports of the group, i.e., $I^{\mathsf{G}}_{\text{in}}$, and the other enclosing the output group ports, i.e., $I^{\mathsf{G}}_{\text{out}}$. Furthermore, each (input and output) interface is divided in subsets according to the enclosed data types of the ports. For example, the group input interface $I^{\mathsf{G}}_{\text{in}}$ contains regular ports ($I^{\mathsf{G}}_{\text{reg\_in}}$), multi-element ports ($I^{\mathsf{G}}_{\text{multi\_in}}$) and the input trigger port ($p^{\mathsf{G}}_{\text{trigg\_in}}$).

$$\begin{aligned} I^{\mathsf{G}} &= I^{\mathsf{G}}_{\text{in}} \cup I^{\mathsf{G}}_{\text{out}}, \text{ where} \\ I^{\mathsf{G}}_{\text{in}} &= I^{\mathsf{G}}_{\text{reg\_in}} \cup I^{\mathsf{G}}_{\text{multi\_in}} \cup \{p^{\mathsf{G}}_{\text{trigg\_in}}\} \text{ and} \\ I^{\mathsf{G}}_{\text{out}} &= I^{\mathsf{G}}_{\text{reg\_out}} \cup I^{\mathsf{G}}_{\text{multi\_out}} \cup \{p^{\mathsf{G}}_{\text{trigg\_out}}\} \end{aligned}$$

4.5.2 Group identification

This section presents the algorithm that identifies the flexible groups in a given component-based system. We see the system as a directed graph, were each component is a node and the trigger port connection between two components is a directed vertex. In this context, identifying the groups is similar to a depth-first search algorithm.

The algorithm, described in Algorithm 1, starts by initializing the main variables. For instance, Γ, representing a set containing all identified groups in the system, is initially an empty set. In Rubus, the clock elements are the system elements that initiate component triggering. Therefore, the algorithm starts traversing the system from the clock elements, by calling the main loop (i.e., the *Top* function) for each triggered clock component. Initially, when starting to traverse the system and there is no formed group, for each encountered and not visited flexible component, a new flexible group with the same hardware allocation is created. The component is added to the created group and the group is added to Γ. The Top function is recursively executed for all of the component's triggering elements (e.g., regular component, flexible component). In order to include all connected flexible components (with the same hardware

```
$\Gamma \leftarrow \emptyset$
$Visited \leftarrow \emptyset$
foreach clock $C$ in the system do
    Top($C$, $NULL$)
end
Top($C$, $G$):
if $C \notin Visited$ then
    add($C$, $Visited$)
    if $flexible(C)$ then
        if $G = NULL \lor alloc(C) \neq alloc(G)$ then
            $G \leftarrow$ createNewGroup()
            $alloc(G) \leftarrow alloc(C)$
            add($G$, $\Gamma$)
        end
        add_last($C$, $G$)
    end
    else
        $G \leftarrow NULL$
    end
    foreach triggering edge $C \rightarrow C'$ do
        Top($C'$, $G$)
    end
end
```

Algorithm 1: Identifying flexible groups

allocation) in the same group, we use a reference to the current group set. This reference is passed to the loop function as a parameter. Whenever a triggered flexible component has a different allocation than the reference group, a new group is created and becomes the current reference group.

4.6 The conversion to Rubus constructs

The conceptual flexible group is realized following the characteristics of a regular component generation, i.e., through an *interface*, *constructor*, *behavior function* and *destructor*, as follows. The generated interface contains all the (input and output) data ports of the group. The constructor generation initializes the resource requirements of the group, e.g., allocates memory space to hold the results from all enclosed components. The group behavior executes the functionalities of the grouped components. The destructor releases the group allocated resources.

Furthermore, we re-wire the connections between the group (input and output) interface and the interfaces of (outside-the-group) components that were initially connected to the group enclosed components.

4.6.1 Generic API

To increase the maintainability of the components resulted from the conversion of flexible components to Rubus constructs and to simplify the generation, we provide a *generic API*. The API abstracts the different characteristics of existent hardware platforms through several functions that transparently call the OpenCL mechanisms that correspond to the utilized platform. There are four functions provided by the API, as follows:

- the *apiCreateBuffer* function allocates GPU memory buffers,
- the *apiReleaseBuffer* function deallocates existing GPU memory buffers,
- the *apiTransferBuffer* function transfers data between different memory systems, and

- *apiSetKernelArg* to set up the parameters for GPU functions.

To illustrate the construction of the API, we present the *apiCreateBuffer* function Listing 4.2 and the *apiSetKernelArg* function Listing 4.3, as follows.

The *apiCreateBuffer* function inspects the current OpenCL version (existing on the platform) to determine which mechanism to request. For the 1.1 and 1.2 OpenCL versions (line 3) that correspond to a platform with distinct CPU and GPU address spaces, the *clCreateBuffer* mechanism is utilized to create an object directly in the GPU address space (line 5). For more technological advanced platforms that support 2.0 and 2.1 OpenCL versions (line 8), it verifies the hardware capabilities, i.e., if it has a full shared address space or shared virtual memory. Based on the finding, it invokes the right mechanism (i.e., *malloc* or *clSVMAlloc*).

Listing 4.2: The *apiCreateBuffer* function

```
1  void *apiCreateBuffer(cl_context context, cl_mem_flags flags, size_t
        size, void *host_ptr, cl_int *errcode_ret)
2  { // Create memory buffer on the device
3  #if !defined(CL_VERSION_2_0) && ( defined(CL_VERSION_1_2) || defined(
        CL_VERSION_1_1) )
4          //Distinct memory allocation buffer";
5          return ((void *)clCreateBuffer(context, flags, size, host_ptr,
               errcode_ret));
6  #endif
7
8  #if defined(CL_VERSION_2_0) && defined(CL_VERSION_2_1)
9          //shared virtual memory
10         cl_device_svm_capabilities caps;
11
12         cl_int err_svm = clGetDeviceInfo(deviceID,
                CL_DEVICE_SVM_CAPABILITIES,sizeof(
                cl_device_svm_capabilities),&caps,0);
13         if (err_svm == CL_SUCCESS && (caps &
                CL_DEVICE_SVM_FINE_GRAIN_SYSTEM) && (caps &
                CL_DEVICE_SVM_ATOMICS) ) {
14                 // Fine-grained system with atomics
15                 return malloc(size);
16         }
17         else if (err_svm == CL_SUCCESS && (caps &
                CL_DEVICE_SVM_FINE_GRAIN_SYSTEM) ) {
```

```
            // Fine-grained system
            return malloc(size);
            }
        else if (err_svm == CL_SUCCESS && (caps &
             CL_DEVICE_SVM_FINE_GRAIN_BUFFER) && (caps &
             CL_DEVICE_SVM_ATOMICS) ) {
            // Fine-grained buffer with atomics
            return clSVMAlloc(context, flags |
                 CL_MEM_SVM_FINE_GRAIN_BUFFER | CL_MEM_SVM_ATOMICS,
                 size, 0);
        }
        else if (err_svm == CL_SUCCESS && (caps &
             CL_DEVICE_SVM_FINE_GRAIN_BUFFER) ) {
            // Fine-grained buffer
            return clSVMAlloc(context, flags |
                 CL_MEM_SVM_FINE_GRAIN_BUFFER, size, 0);
        }
        else if (err_svm == CL_SUCCESS && (caps &
             CL_DEVICE_SVM_COARSE_GRAIN) ) {
            // Coarse-grained buffer
            return clSVMAlloc(context, flags, size, unsigned int
                 alignment);
        }
        else if ( err_svm == CL_INVALID_VALUE ) {
            // No shared-virtual memory
            return ( clCreateBuffer(context, flags, size, host_ptr,
                 errcode_ret) );
        }
#endif
}
```

The *apiSetKernelArg* function, presented in Listing 4.3, abstracts the different functions that are required to set the kernel arguments, on different type of GPU architectures. For example, while for platforms with distinct memory systems, the *clSetKernelArg* function is needed, on platforms with full shared memory, *clSetKernelArgSVMPointer* is required.

Listing 4.3: The *apiSetKernelArg* function

```
cl_int apiSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t
     arg_size, const void *arg_value)
{ // Set the argument of the kernel
```

```

#if !defined(CL_VERSION_2_0) && ( defined(CL_VERSION_1_2) || defined(
     CL_VERSION_1_1) )
      // Distinct memory systems
      return (clSetKernelArg(kernel, arg_index, arg_size, arg_value));
#endif

#if defined(CL_VERSION_2_0) || defined(CL_VERSION_2_1)

      cl_device_svm_capabilities caps;
      cl_int err_svm = clGetDeviceInfo(deviceID,
           CL_DEVICE_SVM_CAPABILITIES,sizeof(cl_device_svm_capabilities
           ),&caps,0);

      if ( err_svm == CL_SUCCESS && ( (caps &
           CL_DEVICE_SVM_FINE_GRAIN_SYSTEM) || ((caps &
           CL_DEVICE_SVM_FINE_GRAIN_BUFFER)) || ((caps &
           CL_DEVICE_SVM_COARSE_GRAIN)) )
                  return clSetKernelArgSVMPointer(kernel, arg_index,
                       arg_value);
      else return (clSetKernelArg(kernel, arg_index, arg_size,
           arg_value));
#endif
}
```

All other API functions are developed in the same manner, i.e., inspecting the OpenCL version existing and platform characteristics, and calling the corresponding mechanisms.

4.6.2 Code generation

During the realization step, the system elements (e.g., regular components, groups) are converted into source-code. As we introduced the concept of component group, this section presents the realization of a group through its parts, that is:

- the interface – generated as a structure that contains the (input and output) data ports,
- the constructor – generated as a function in which resources are allo-

cated,

- the behavior function – generated as function, and
- the destructor – generated as a function in which the resources are released.

Using the existing way that Rubus defines the interface of a regular component, we generate the *group interface* in a similar manner, as presented in Listing 4.4. The *SWC_Group_iArgs* interface is defined as a structure (lines 35-38) with two elements corresponding to the output and input interfaces. The output interface *OP_SWC_iArgs* is constructed as a structure (lines 28-32), where the elements are the data ports of the group output interface $I^{\mathsf{G}}_{\mathtt{out}}$. Similarly, *IP_SWC_iArgs* is a structure that encloses the data ports of the group input interface $I^{\mathsf{G}}_{\mathtt{in}}$.

Besides input data ports, *IP_SWC_iArgs* interface contains the so-called *configuration* ports. Each flexible component is equipped with a configuration interface. Through it, the system designer provides appropriate settings regarding the number of device threads used to execute the functionality. For example, a flexible component allocated on GPU could receive, through the configuration interface, settings to use 2048 GPU threads. In the flexible component realization, the configuration interface is generated as regular input data port in order to not introduce additional Rubus framework elements. In our generation, we use the same rational, i.e, the flexible group is equipped with a configuration interface generated as an input data port, for each enclosed component (line 22).

The settings received through the configuration interface are inserted in the *GPU_settings* structure. The first four elements (lines 5-8) refers to the number of device-threads used by the functionality, while the rest of the elements (lines 9-11) are settings related to the environment, such as the *command_queue* mechanism.

Listing 4.4: Interface code

```
/* device-settings for each flexible component C */
<counter = 1>
<foreach $C$ in $G$>
typedef struct {
      int blockDim_x;
      int blockDim_y;
      int gridDim_x;
      int gridDim_y;
      cl_context context;
      cl_command_queue cmd_queue;
      cl_device_id device_id;
 }settings<counter+=1>;
<endforeach>

/* the group input ports */
<counter = 1>
typedef struct {
<foreach p in $I_{in}^{G}$>
      <p.type> *<p.name>;
<endforeach>
<foreach $C$ in $G$>
      settings<counter> *cfg<counter>;
      <counter += 1>
<endforeach>
}IP_SWC_iArgs;

/* the group output ports */
typedef struct {
<foreach p in $I_{out}^{G}$>
   <p.type> <p.name>;
<endforeach>
}OP_SWC_iArgs;

/* the interface of the group */
typedef struct {
  IP_SWC_iArgs IP;
  OP_SWC_iArgs *OP;
}SWC_Group_iArgs;
```

The *constructor*, illustrated in Listing 4.5, encloses all the information regarding the group initialization, as follows. The listing starts by allocating memory for each flexible component from the group. That data received by a component through the input ports is the input data for the functionality, while the functionality outcomes are sent through the output ports. Therefore, corresponding to each output data port, we allocate memory to hold the functionality results. Due to the specifics of the OpenCL, a kernel function must store a regular output value (e.g., integer value) in a one-value memory buffer. Thus, we allocate one-value memory buffers for regular output ports (line 4). For data of m-elem type, the memory buffer is allocated with an appropriate size (line 8). Moreover, in line 15, the multi-element ports that are considered output ports of the group are linked to the corresponding memory locations. This is done because these ports will be wired to outside-of-the-group ports, and the system communication will be accomplished by using the values of the connected ports.

The core part of the constructor defines the group functionality. A string variable encloses the functionalities of the grouped components, i.e., the kernel function name (line 21), the arguments (lines 23, 26, 29 and 32) that correspond to the input and output component ports, and the component functionality (line 37). The string variable is loaded into a program object (line 43) and then compiled to create a dynamic library (line 46). In the last part of the constructor, kernel objects are constructed for all flexible components (line 51), alongside with the individual settings regarding the number of used device-threads (line 54 and 55). We mention that these settings are provided by the system designer, using the configuration interface port.

The reason to include the group functionality inside the constructor is to create the dynamic library once (i.e., by creating and compiling the *program* object), at the system initialization stage.

Listing 4.5: Constructor code

```
1  /* create memory buffers for each flexible component that is part of a
        flexible group */
2  <foreach $C$ in $G$>
3   <foreach p in $I^{C}_{reg\_out}$>
4     void *result_<p.name> = apiCreateBuffer(settings->contex,
         CL_MEM_WRITE_ONLY, sizeof(<p.type>),NULL,NULL);
```

```
 <endforeach>

 <foreach p in $I^{C}_{multi\_out}$>
  void *result_<p.name> = apiCreateBuffer(settings->contex,
      CL_MEM_WRITE_ONLY, <p.width*p.height*p.size>,NULL,NULL);
 <endforeach>
<endforeach>

/* connect the output ports of the group with the created memory
     buffers */
<foreach $C$ in $G$>
 <foreach p in $I^{G}_{out}$>
  <p.name>->data = (unsigned char*) result_<p.name>;
 <endforeach>

 const char *source_string ="
<counter_kernel = 1>
<foreach $C$ in $G$>
  __kernel void flexible_kernel<counter_kernel>(
   <foreach $p$ in $I^{C}_{reg\_in}$>
    <p.type> <p.name>,
   <endforeach>
   <foreach $p$ in $I^{C}_{reg\_out}$>
    __global <p.type> *result_<p.name>,
   <endforeach>
   <foreach $p$ in $I^{C}_{multi\_in}$>
    __global <p.type> *<p.name>,
   <endforeach>
   <foreach $p$ in $I^{C}_{multi\_out}$>
    __global unsigned char *result_<p.name>,
   <endforeach>
   ){

/* flexible component functionality */
<$F^{C}$>
}";
<counter_kernel += 1>
<endforeach>

/* Create a program from the kernel sources */
  cl_program program = clCreateProgramWithSource(settings->context, 1,
      (const char **)&source_string, NULL, NULL);

```

```
/* Build the program */
  clBuildProgram(program,1,&(settings->device_id), NULL, NULL, NULL);

<counter_kernel=1>
<foreach C in G>
/* Create the kernel object */
  cl_kernel kernel<counter_kernel> = clCreateKernel(program, "
       flexible_kernel<counter+=1>", NULL);

/* individual settings - device threads usage */
  int total_thrd<counter_kernel>[2] = {(settings->gridDim_x),(settings
       ->gridDim_y)};
  int group_thrd<counter_kernel>[2] = {(settings->blockDim_x), (
       settings->blockDim_y};
  <counter_kernel+= 1>
<endforeach>
```

The execution of the group functionality is enclosed in the *behavior function* (Listing 4.6) which is performed every time the group is activated. To execute the functionality using the OpenCL model, the host needs to send to the selected device (i.e., CPU or GPU), the execution command of the desired kernel function. However, before triggering the execution, the input data and locations for output results need to be specified. Hence, the first part of the behavior function handles the parameters (i.e., provide the values) of the group enclosed kernels. Basically, the parameters of a kernel are the input data and output data location of the corresponding flexible component. For the input ports of the enclosed components that are not considered the group ports, we provide the values received from the connected ports by using the defined *connect* construct. This is done by directly providing the allocated memory location corresponding to the connected ports (lines 11 and 25). In this way, the communication between kernel functions of different connected components is directly realized inside the group, at the functionality level.

For (regular and multi-element) output ports, we provide the data existing in the corresponding allocated memory (lines 16 and 31). Based on the order of the grouped set, the functionalities (i.e., the *kernel* objects) of the enclosed components are triggered to be executed on the selected hardware (line 39). In the last part, we copy the computed one-value of the allocated memory buffers, to the corresponding regular data output ports of the group (line 46). When the wiring between existing system components and groups will be done, the regular output ports of the group will provide a regular data (e.g., integer value) instead of a (one-value) memory buffer (i.e., pointer). In this way, the Rubus rules that realizes communication between data ports are not interfered.

Listing 4.6: Behavior function

```
1  /*Set the kernel arguments of each enclosed component*/
2  <counter_kernel = 1>
3  <counter_arg = 0>
4  <foreach $C$ in $G$>
5   <for each $p$ in $I^{C}_{reg\_in}$>
6    /* for regular input ports of flexible components that are considered
         input ports of the group */
7    <if (p in $I^{G}_{reg\_in}$)>
8     apiSetKernelArg(kernel<counter_kernel>,<counter_arg+=1>, sizeof(
          <p.type>), (void*)&<p.name>);
9    /* for regular input ports of flexible components that are not input
        ports of the group */
10   <else>
11    apiSetKernelArg(kernel<counter_kernel>,<counter_arg+=1>, sizeof(
          <p.type>), (void*)&result_<(p.connect).name>);
12   <endif>
13    <counter_kernel+= 1>
14  <endforeach>
15  <for each $p$ in $I^{C}_{reg\_out}$>
16    apiSetKernelArg(kernel<counter_kernel>,<counter_arg>, sizeof(
         <p.type>), (void*)&result_<p.name>);
17    <counter+= 1>
18  <endforeach>
19  <foreach $p$ in $I^{C}_{multi\_in}$>
20  /* multi-element input ports of flexible components that are input
        ports of the group */
21   <if (p in $I^{G}_{reg\_in}$)>
22    apiSetKernelArg(kernel<counter_kernel>,<counter_arg>,
          <p.width*p.height*p.size>, (void*)&<p.name>);
23   /* multi-element input ports of flexible components that are not
        input ports of the group */
24   <else>
25    apiSetKernelArg(kernel<counter_kernel>,<counter_arg>,
          <p.width*p.height*p.size>, (void*)&result_<(p.connect).name>);
```

```
26    <endif>
27
28     <counter+= 1>
29   <endforeach>
30   <for each $p$ in $I^{C}_{multi\_out}$>
31     apiSetKernelArg(kernel<counter_kernel>,<counter_arg>,
           <p.width*p.height*p.size>, (void*)&result_<p.name>);
32   <endforeach>
33   <counter_kernel+=1>
34  <endforeach>
35
36  /* Execute the OpenCL kernels of the flexible components */
37  <counter=1>
38  <foreach $C$ in $G$>
39    clEnqueueNDRangeKernel(settings->cmd_queue, kernel<counter>, 2, NULL
          , total_thrd<counter>, group_thrd<counter>, 0, NULL, NULL);
40    <counter+=1>
41  <endforeach>
42
43  /* copy the regular output(s) to the corresponding regular output port
         (s) of the group */
44  <foreach $C$ in $G$>
45   <foreach p in $I^{G}_{reg\_out}$>
46    apiEnqueueReadBuffer(settings->cmd_queue, result_<p.name>, CL_TRUE,
          0, sizeof(<p.type>), &<p.name>, 0, NULL, NULL);
47   <endforeach>
48  <endforeach>
```

The *destructor* releases the resources allocated by the constructor. Basically, the kernel objects (line 4), the program object (line 7) and the allocated memory buffers (line 10) are released.

Listing 4.7: Destructor code

```
1  /* Clean up */
2  <counter_kernel = 1>
3  <foreach $C$ in $G$>
4   clReleaseKernel(kernel<counter_kernel>);
5   <counter_kernel+=1>
6  <endforeach>
7   clReleaseProgram(program);
8  <foreach $C$ in $G$>
9   <foreach p in $I^{C}_{out}$>
```

```
    apiReleaseBuffer(result_<p.Name>);
  <endforeach>
<endforeach>
```

4.6.3 Connection rewiring

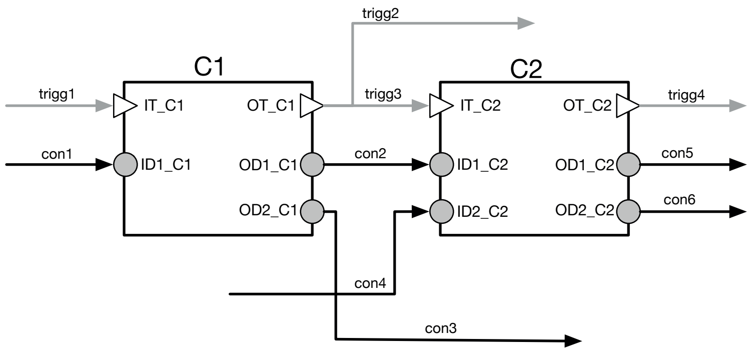
The way Rubus framework models the wiring between the system components needs to be partially changed by the fact that several components are grouped as a single component. Inside the group, the connections that wire the enclosed components are ignored because the data communication is directly accomplished now at the functionality (level lines 11 and 25 in Listing 4.6). The connections between the group's components and everything outside the group needs to be modified in the following way. Instead of constructing the wiring between the interfaces of the outside-the-group components and the interfaces of the enclosed components, we re-wire the connections between the interfaces of the outside components and the group.

The trigger connection leading in to the group is handled in the same way as the data connections, but outgoing trigger connections should all be rewired to originate from the single output trigger port of the group interface.

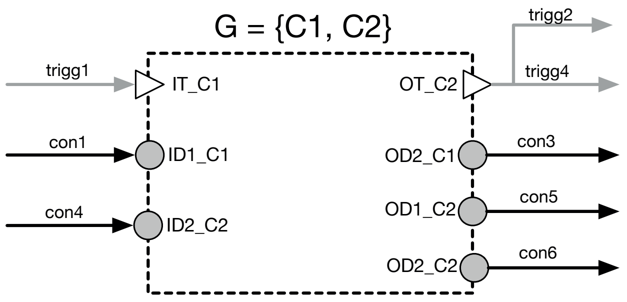
Thus, when considering a connection from port p_1 of component C_1 to port p_2 of component C_2 (i.e., where $p_1 \in I^{\mathsf{C}_1}$ and $p_2 \in I^{\mathsf{C}_2}$), three cases must be considered:

- If $\exists G : C_1 \in G \wedge C_2 \in G$, then the connection is removed.
- If $\exists G : C_1 \in G \wedge C_2 \notin G$, there are two sub cases:
  - If p_1 and p_2 are data ports, then the connection remains.
  - If p_1 and p_2 are trigger ports, then the connection should be rewired from the output trigger port of the group $p^{\mathsf{G}}_{\mathrm{trigg\_out}}$ to p_2 .
- If $\exists G : C_1 \notin G \wedge C_2 \in G$, then there are two cases:
  - If p_1 and p_2 are data ports, then the connection remains.

– If p_1 and p_2 are trigger ports, then the connection should be rewired from p_1 to the input trigger port of the group $p^{\mathsf{G}}_{\mathrm{trigg\_in}}$.



(a) Two connected flexible components



(b) A group and its connections

Figure 4.5: The rewiring of a group ports

The connection rewiring rules are described through an example presented by Figure 4.5. Two connected flexible components $C1$ and $C1$, and their connections are presented in Figure 4.5a. $C1$ has one input data port $ID1\_C1$ and two output data ports, where $OD1\_C1$ is connected to $C2$ and $OD2\_C1$ is connected to another component. Furthermore, its output trigger port OT_C1 triggers, besides $C2$, another component. Assuming that both $C1$ and $C1$ are allocated onto the same processing unit, their realization is optimization via a group G presented in Figure 4.5b. The group inherits $ID1\_C1$ and $ID2\_C2$ as input data ports, and $OD2\_C1$, $OD1\_C2$ and $OD1\_C2$ as output data ports. The connections to the group data ports remain unchanged. Regarding the group trigger ports, the trigger connections *trigg1* is rewired to the input trigger port, while the output trigger port is rewired with $trigg2$ and $trigg4$.

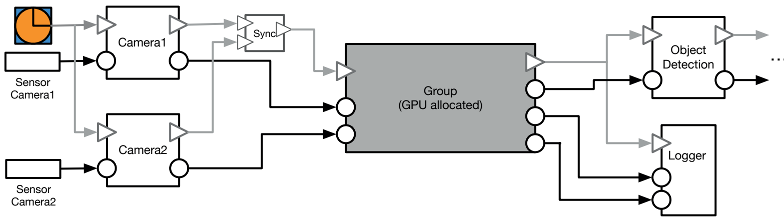


Figure 4.6: A group and its connections in the vision system

In the vision system case study, assuming that all flexible components were allocated on the GPU, a group is constructed as presented in Figure 4.6. The group inherits two input data ports, both from *MergeAndEnhance* component, and three out output data ports from *EdgeDetection*, *CompressGrayscale* and *CompressRGB* respectively. The connections of the inherited data ports remain, the data connections between the enclosed components are removed, and new connections are rewired to the trigger ports of the group.

4.7 Component communication support

A concept that complements the flexible group by letting it to be (re-)used with no changes on different platforms, is the automatically generated *adapter*. When constructing regular components with GPU capabilities, different (data transfer) operations are required to be encapsulated inside the components. We automatically externalize these required operations in artifacts that transparently realize the communication between the groups and components of the system. In this way, the group becomes platform-independent.

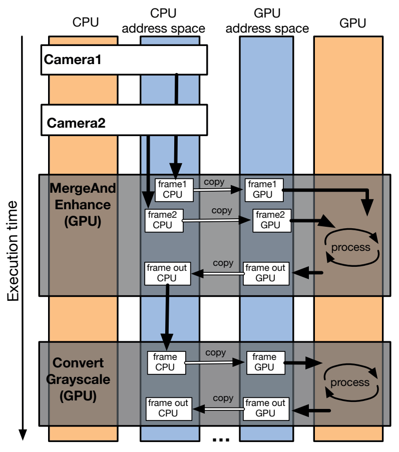
4.7.1 Adapter identification

Components with GPU capability require, for particular platforms, to encapsulate operations that transfer data between memory systems. To exemplify, we introduce Figure 4.7 which describes the execution of (a part of) the vision system on three different type of platforms. The vision system is composed of regular components (e.g.*Camera1*, *Camera2*) and components with GPU capability (e.g., *MergeAndEnhace*, *ConvertGrayscale*). For platforms with distinct CPU and GPU memory spaces, illustrated in Figure 4.7(a), all components, including the ones with GPU capability, are aware only of the main (CPU) memory system. Therefore, *MergeAndEnhance* has encapsulated activities to transfer the two data frames from the CPU to the GPU memory system, and the other way around, i.e., the output from the GPU back to the main (CPU) memory space. In a similar way, *ConvertGrayscale* component encapsulates transfer data activities between the CPU and GPU memory systems. For the second type of platform, where there is a shared virtual memory layer, similar activities are encapsulated inside the components with GPU capability as depicted by Figure 4.7(b). In this case, there is no need for activities to transfer data from SVM to main memory, due to the characteristics of the platform, i.e., components, either regular or with GPU capability, can access directly SVM. The last type of platform, where the CPU and GPU share the same memory, no activity is needed to be encapsulated inside *ImageAndEnhance* and *ConvertGrayscale*. In this case, all components directly access the (same) memory.

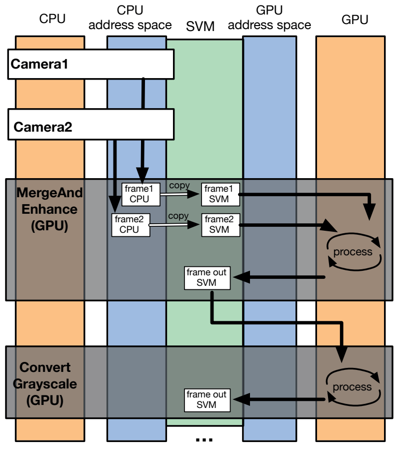
Instead of being encapsulated inside the components with GPU capability, the transfer activities are lifted from the component level and automatically provided through transparent adapters, based on the platform specifications.

We introduce two types of adapters corresponding to the transfer activity directions, i.e., *inAdapter* and *outAdapter* adapters, each one accomplishing either:

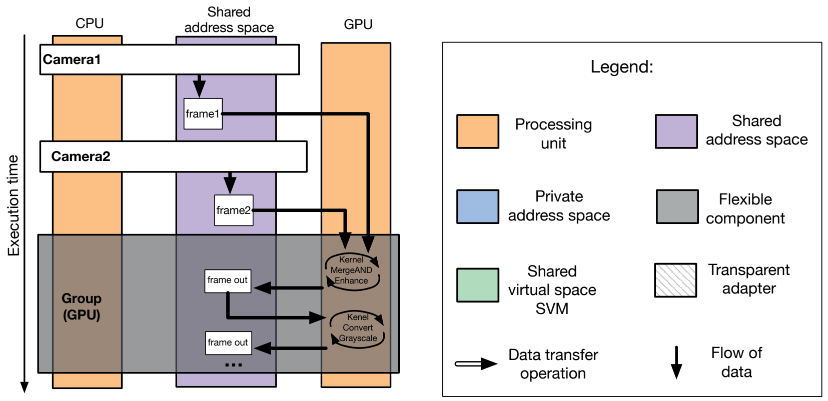
- a one-to-one port communication between two components, or
- a one-to-many port communication between several components.



(a) Distinct address spaces



(b) Shared virtual memory



(c) Shared address space

Figure 4.7: Component activities on three platforms

In our approach, adapters are implemented as regular components by following the component model specification, and are automatically generated in two steps: *I)* introduce the adapters, and *II)* merge the adapters where possible.

The steps are presented as follows.

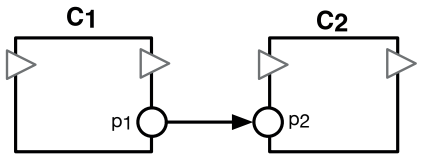


Figure 4.8: A data connection between two components

Step I. The following rules, using a simple example presented in Figure 4.8, define the introduction of adapters and their port connections as follows:

a) Two components C_1 and C_2 communicate through p_1 and p_2 data ports; both ports are of the multi-element type.

b) One of the components is allocated on the GPU and the other component is regular (i.e., executed by the CPU).

- If the platform has a full-shared memory architecture, then no adapter is needed.
- If the platform has a shared virtual memory, then:
  - if C_1 is regular and C_2 is executed by the GPU, than an *inAdapter* adapter is needed; it has one input port connected to p_1 and one output port connected to p_2. The connected data ports have the same type.
  - if C_1 and C_2 are allocated on the GPU, then no adapter is needed.
- If the platform has CPU and GPU distinct memories, then:
  - if C_1 is regular, than a *inAdapter* adapter is needed; it has one input port connected to p_1 and one output port connected to p_2.

– if C_1 is allocated on the GPU and C_2 is regular, than an *outAdapter* adapter is needed; it has one input port connected to p_1 and one output port connected to p_2.

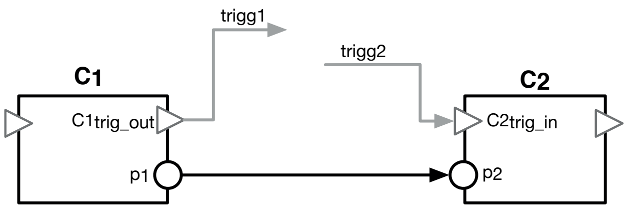


Figure 4.9: Two connected components

Regarding the trigger ports of an adapter, their connections are constructed as following. Using an example as presented in Figure 4.9, let be C_1 and C_2 two components allocated on different processing units. Using the rules from Step I, and assuming an adapter is needed, its trigger ports $p_{\mathsf{trig\_in}}$ and $p_{\mathsf{trig\_out}}$ are connected as following:

- the input trigger port $p_{\mathsf{trig\_in}}$ is connected to $trigg2$, and
- the output trigger port $p_{\mathsf{trig\_out}}$ is connected to the input trigger $C2_{\mathsf{trig\_in}}$.

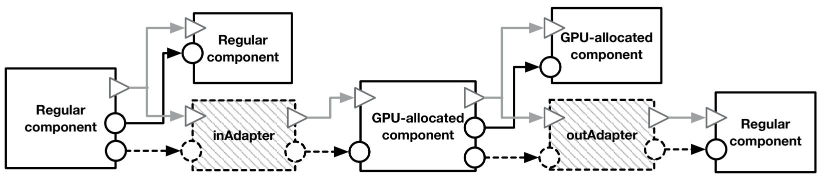


Figure 4.10: Basic generation of adapters

We exemplify the generation of adapters in Figure 4.10, where an *inAdapter* adapter facilitates the communication between regular and GPU-allocated flexible components. Similarly, an *outAdapter* adapter facilitates communication between GPU-allocated and regular components. No adapters are required when connected components are allocated on the same processing units.

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

- when a regular component communicates with several GPU-allocated flexible components through one output port, the communications are done through a single *inAdapter* adapter.
- when a GPU-allocated flexible component communicates with several regular components through one output port, the communications are done through a single *outAdapter* adapter.

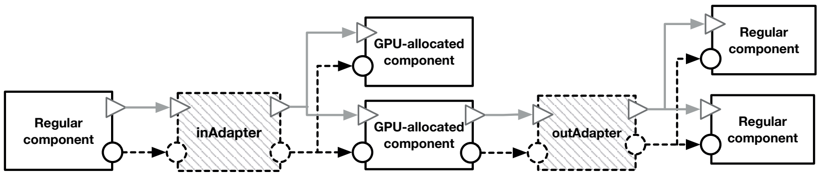
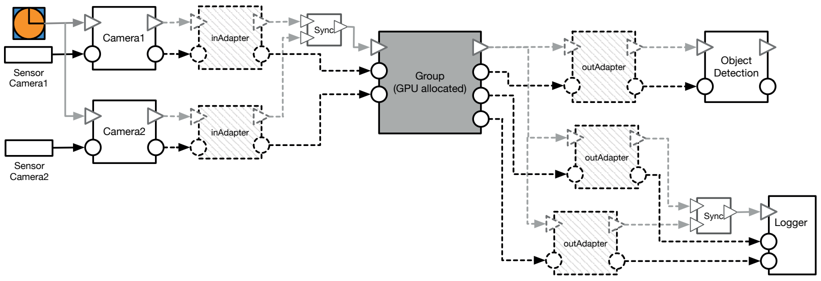


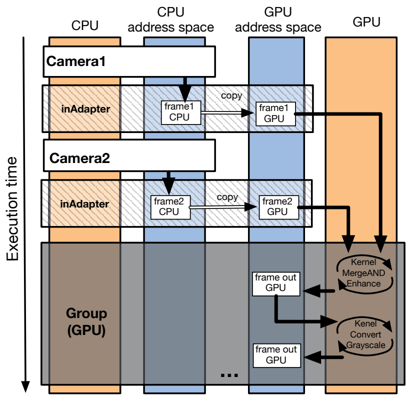
Figure 4.11: Optimization of generation of adapters

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. Figure 4.11 exemplifies the optimization step, where one *inAdapter* adapter transfers data on the GPU memory system and provides it to two GPU-allocated flexible components. Similarly, a single *outAdapter* adapter copies once data on main memory system.

We exemplify the adapter concept via the vision system, when the application is executed on three different platforms. In Figure 4.12, two views of the vision system are presented when executed on a platform with CPU and GPU distinct memory systems. In the first view, Figure 4.12(a) presents the generation of all the required adapters for this type of the platform, i.e., two *inAdapter* adapters, and three *outAdapter* adapters. The activities of the vision system is described in Figure 4.12(b), where the *inAdapter* adapters transfer



(a) Generated adapters for a platform with distinct memory addresses

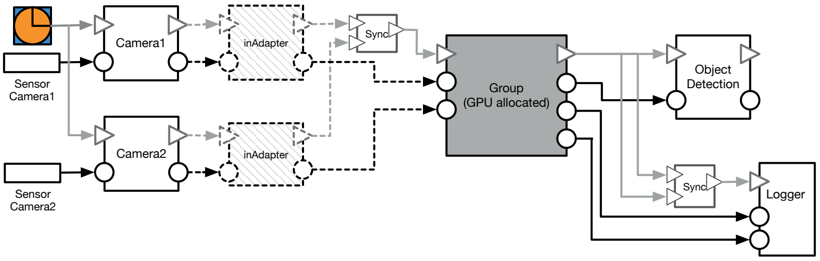


(b) Distinct address spaces

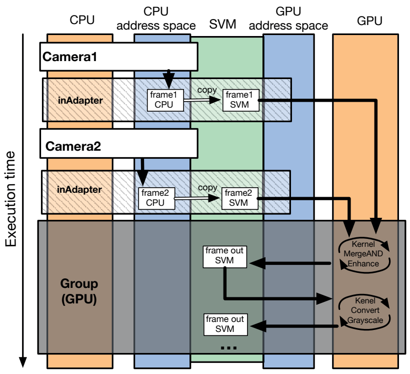
Figure 4.12: Adapter generation on different type of platforms

the data from the CPU to the GPU memory system, and then, directly accessed from the kernel of the enclosed *MergeAndEnhance* component.

Similarly, for platforms with shared virtual memory support, Figure 4.13 presents the two views of the vision system. In Figure 4.13(a), the vision system is enriched only with *inAdapter* adapters. For this particular platform, there is no need for *outAdapter* adapters because all components can directly access the SVM space. The activities are presented in Figure 4.13(b) where the two *inAdapter* adapters transfer the data on the SVM memory, from where is



(a) Generated adapters for a platform with virtual shared memory address



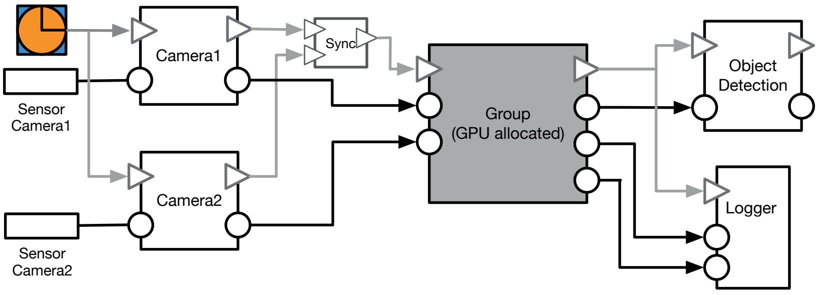
(b) Distinct address spaces

Figure 4.13: Adapter generation on different type of platforms

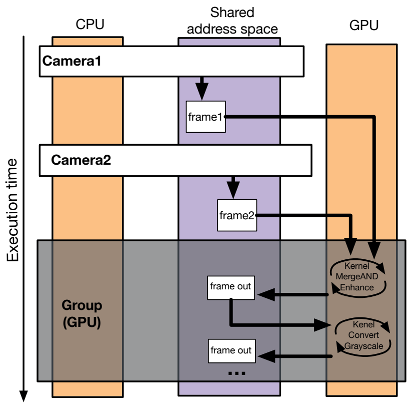
accessed by the kernel of *MergeAndEnhance*.

In the last case, where the platform has full shared memory, Figure 4.14 presents the two views for the vision system. In this case, there is no generation of adapters; all regular and/or GPU-allocate components can directly access the memory.

We mention that, for the vision system, we did not apply the step II of the adapters generation because there were no situations to optimize them.



(a) Generated adapters for a platform with distinct memory addresses



(b) Distinct address spaces

Figure 4.14: Adapter generation on different type of platforms

Listing 4.9: 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);
```

4.7.2 Adapter realization

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 device (for *inAdapter* adapters) or main memory (for *outAdapter* adapters). The output port is linked to the location that holds the copied input data (line 5).

Listing 4.8: 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.

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 4.10: Destructor code of adapters

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

4.8 Evaluation

For the evaluation, we applied our approach on the vision system example. The vision system, as described by Figure 4.15, contains five components that are suitable to be executed on the GPU.

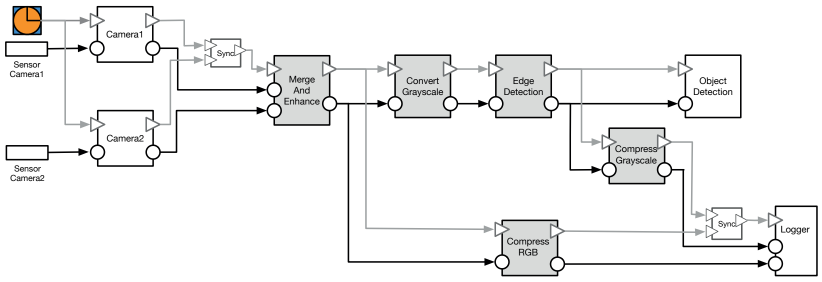


Figure 4.15: The Rubus design of the vision system

To evaluate our approach, we constructed five allocation scenarios. Given that only five components may benefit from the GPU execution, Table 4.1 describes their allocation in all five scenarios. While in Scenario 1 all five components are allocated to the GPU, in Scenario 5 we allocate the components to the CPU. For the rest of the scenarios, we alternate in allocating the components to CPU and GPU.

For each of the constructed scenarios, we implemented three versions of the vision system, as follows. In the naïve version (V1), each of the five components is constructed as a regular component that encapsulates all the required GPU information. In the second version (V2), the five components are seen as flexible components which are realized as regular components with their corresponding allocation. For the third version (V3), the flexible components are optimized through flexible groups, before being converted into regular compo-

Table 4.1: Allocations scenarios for the vision system

Flexible Component	**Hardware allocation scenario**				
	1	**2**	**3**	**4**	**5**
MergeAnd Enhance	GPU	GPU	GPU	GPU	CPU
Convert Grayscale	GPU	GPU	GPU	CPU	CPU
Edge Detection	GPU	CPU	GPU	GPU	CPU
Compress RGB	GPU	GPU	CPU	CPU	CPU
Compress Grayscale	GPU	CPU	CPU	CPU	CPU

nents. Generated adapters are used for V2 and V3 versions. For each scenario, were used three different hardware platforms that contain GPUs: *a)* a PC with an NVIDIA dGPU architecture, *b)* an embedded platform with an AMD Kabini SoC with shared virtual memory architecture[1], and *c)* an embedded platform with an AMD Carrizo SoC with full shared memory architecture[1].

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

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

The naïve version is constructed with components that hard-code all the required GPU information. To exemplify, we present in more details the construction of the *ConvertGrayscale* component. Listing 4.11 illustrates the code

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

that is manually written to construct the behavior function of the *ConvertGrayscale* component.

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

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

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

- The *result* buffer is created (line 51) on the device (e.g., GPU) to hold the data resulted from the kernel execution.

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

Listing 4.11: The code encapsulated by *ConvertGrayscale*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

/* Set the arguments of the kernel */
SetKernelArg(krn,0,sizeof(cl_mem),(void*)&input);
SetKernelArg(krn,1,sizeof(int),(void *)&(args->IP.ID_input1->width));
SetKernelArg(krn,2,sizeof(int),(void *)&(args->IP.ID_input1->height));
SetKernelArg(krn,3,sizeof(cl_mem),(void *)&result);

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

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

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

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

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

Table 4.2 describes the created groups and generated adapters for each considered scenario and platform. In Scenario 1, when all flexible components are allocated on the GPU, depending on the used platform, there are different numbers of generated adapters. For the platform with distinct memory systems (dGPU), there are two *inAdapters* and three *outAdapters* created for V2 and V3 versions as illustrated by Figure 4.16. In Figure 4.16(a), we present the generated adapters for a platform with distinct memory systems, considering the flexible component-based version (V2). The two generated *inAdapters* transfer data from the host (CPU) to the device (GPU) and provide it to the *MergeAndEnhance* component. The three generated *outAdapters* transfer data from the device to the host as follows:

- the first adapter communicates data between *EdgeDetection* and *ObjectDetection* components,
- the second adapter transfers the compressed gray scale frame onto the host memory and provides it to the *Logger* component, and
- the last adapter communicates the compressed color frame to the *Logger* component.

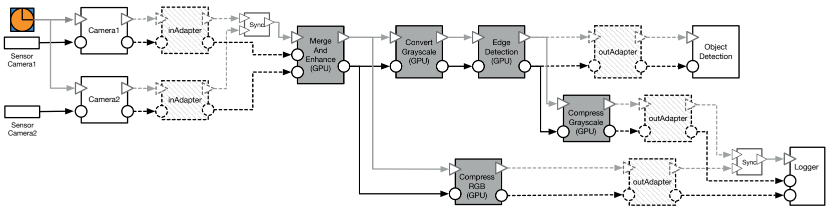
Figure 4.16(b) presents the flexible group-based version (V3) for the same type

Table 4.2: The number of adapters and formed groups

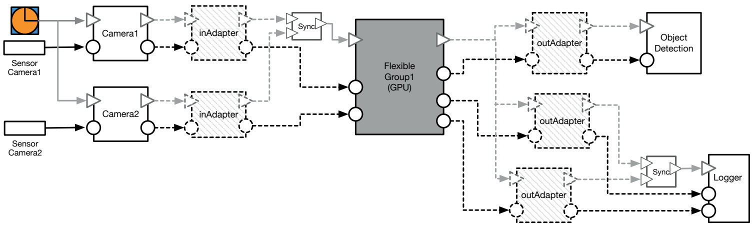
Scenario	**Platform type**	**Number of adapters**						**Number of groups**
		inAdapter			**outAdapter**			
		V1	**V2**	**V3**	**V1**	**V2**	**V3**	**V3**
1	dGPU	0	2	2	0	3	3	1
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
2	dGPU	0	2	2	0	3	3	2
	iGPU1	0	2	2	0	0	0	
	iGPU2	0	0	0	0	0	0	
3	dGPU	0	2	2	0	3	3	3
	iGPU1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

dGPU - CPU and GPU distinct memory platform
iGPU1 - Shared Virtual Memory platform
iGPU2 - Shared Memory platform
V1 - Naïve version
V2 - Flexible component-based version
V3 - Flexible group-based version

of platform, where one group is created with two input ports and three output ports. The same number of adapters are created, connecting the group with the rest of the system.



(a) Generated adapters



(b) Created group

Figure 4.16: Adapter generation and group creation for platforms with distinct memories

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

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

Table 4.3 presents the generated and manually written code for all five scenarios, platforms and system versions. Regarding the manually written characters, the naïve version contains the highest number, with a maximum of 14909 characters for the dGPU type of platform. While the characteristics of the platform change, the manual written code is changing given that e.g., there are different requirements for data transfer activities. For platforms with virtual memory (iGPU1), where there is no need for transferring data from device to host, there are written 13474 characters. For the last type of platform with a share memory system (iGPU2), due to the fact that there is no need of data transfer activities, a number of 11961 characters are required.

Regarding the generated characters, the number of generated adapters and groups (see Table 4.2) influence the generated code size of the vision system. We notice that for shared memory platforms (dGPU), the flexible component-based version (V2) has 9007 generated characters that includes two *inAdapters* and three *outAdapters*. For the same system version, in the case of the platform with virtual memory (iGPU1), the number of generated characters drops to 6852 because there are no *outAdapters*. Finally, for shared memory platform (iGPU2) where there are no required adapters, there are 5414 generated characters. In the case of the flexible group-based version (V3), there are less characters generated than V2 version due to the enclosing of components into group entities. For all scenarios and platforms, there are no generated characters for the naïve version as all the code is manually written.

For the second part of the evaluation, Table 4.4 presents the execution time for all three versions of the system, in all five scenarios. We mention that

Table 4.3: The code size of the system versions

Scenario	**Platform type**	**Code size (# characters)**					
		Generated			**Written**		
		V1	**V2**	**V3**	**V1**	**V2**	**V3**
1	dGPU	0	9007	8816	14909	1720	1720
	iGPU1	0	6852	6759	13474	1720	1720
	iGPU2	0	5414	5223	11961	1720	1720
2	dGPU	0	9007	8953	14909	1720	1720
	iGPU1	0	6852	6796	13474	1720	1720
	iGPU2	0	5414	5360	11961	1720	1720
3	dGPU	0	9007	8945	14904	1720	1720
	iGPU1	0	6852	6788	13474	1720	1720
	iGPU2	0	5414	5352	11961	1720	1720
4	dGPU	0	9007	9007	14904	1720	1720
	iGPU1	0	6852	6852	13474	1720	1720
	iGPU2	0	5414	5414	11961	1720	1720
5	dGPU	0	9007	8816	14909	1720	1720
	iGPU1	0	6852	6759	13474	1720	1720
	iGPU2	0	5414	5223	11961	1720	1720

dGPU - CPU and GPU distinct memory platform
iGPU1 - Shared Virtual Memory platform
iGPU2 - Shared Memory platform
V1 - Naïve version
V2 - Flexible component-based version
V3 - Flexible group-based version

the execution time was computed for the entire system, from the starting of the execution of *Camera1* and *Camera2* until *ObjectDetection* and *Logger* finished their execution. Furthermore, the experiments were done on a machine with 2,6 GHz i7 processor and two distinct memory systems (dGPU type of platform).

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

Table 4.4: The execution time of the system versions

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

V1 - Naïve version
V2 - Flexible component-based version
V3 - Flexible group-based version

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

V

Chapter 5

Allocation optimization

The *flexible component* concept introduced in the previous chapter, facilitates and increases the flexibility of designing embedded systems with GPUs. On the other side, it brings some additional complexity during the system construction by increasing the allocation challenge. In a system composed of many flexible components, it can be difficult to choose a good (or optimal) component allocation in order to match (or optimize) system requirements.

In this chapter, we introduce a semi-automatic method that provides optimal allocation schemes of flexible components. The chapter starts by describing the overview of the allocation method, after which it presents its formal definition. In the last part, we apply the allocation method on the underwater robot example system.

Regarding the research goals and contributions presented in the Introduction, Contribution 4 is addressed in this chapter, that covers Research Goal 3.

5.1 Allocation optimization overview

Once the design of the system is constructed, a challenge appears when determining the allocation of the flexible components with respect to certain optimization goals. Moreover, the challenge is increased by the fact that, at the adapter construction step, adapters are automatically generated for data transfer activities, and these adapters influence the suitability of the allocation due to e.g., their memory usage and energy consumption.

As an alternative to the manual allocation method, we propose a semi-automatic method that computes optimized allocation schemes for a component-based embedded system. For the proposed allocation method, we focus only on a single type of platform, i.e., the platform with a single CPU-GPU chip where each of the processing units has its own memory. As input, the method receives the system model (capturing relevant properties of both software and hardware), and the optimization criteria, described in the following section.

5.2 Allocation 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.

5.2.1 Input

The system model, which represents the input to our allocation method, includes both the software and hardware aspects. Using our introduced development process overview (see Figure 4.3), the software aspects are described via the software design, i.e., connected (regular and flexible) components that follow the pipe-and-filter interaction style, while the platform model provides the hardware aspects.

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 component, such as the variables defined in the behavior function, 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. This property is needed because, 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. 5.1 presents a section of a Rubus application composed of a regular (i.e, *C2*) and three flexible components (i.e., *C1*, *C3* and *C4*). A flexible component is characterized by 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 0.15 Wattseconds; the GPU-related properties are set to zero.

As explained in Section 5.1, the hardware platform is currently limited to a contains a single CPU-GPU chip, where each processing unit has its private memory space. The properties that characterize the hardware platforms are: *i)* the available GPU memory, and *ii)* the available CPU memory.

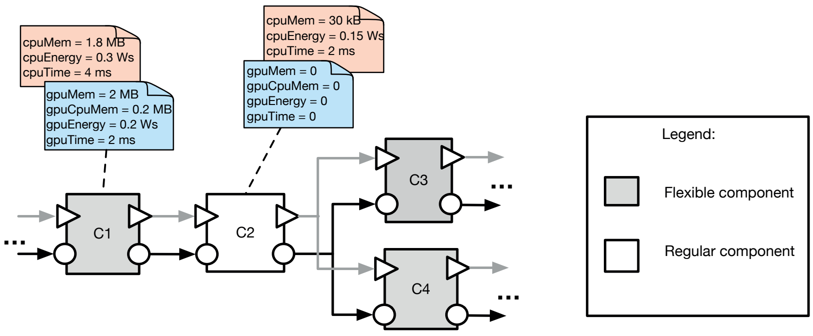


Figure 5.1: Connected components and their (CPU and GPU) properties

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

Moreover, we define the set $S = \{\langle f_1, C_1\rangle, \ldots, \langle f_q, C_q\rangle\}$ describing the triggering of components, where $C_i \subseteq C$ is a subset containing the components that are directly or indirectly 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., $C_1, \ldots, C_q$ 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 *conn*), and eight functions to represent properties of individual components and ports. The definitions of the functions are described by Table 5.1.

Regarding the hardware platform, we introduce two variables to describe the platform characteristics and two variables through which the developer can

Table 5.1: Ten functions that describe 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.
$conn : O \rightarrow 2^{\mathsf{I}}$	$conn(op)$ = the set of input ports connected to op.
$portSize : O \rightarrow \mathbb{N}$	$portSize(op)$ = 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 the CPU.
$gpuTime : C \rightarrow \mathbb{N}$	$gpuTime(c)$ = the execution time of c on the GPU.
$cpuEnergy : C \rightarrow \mathbb{N}$	$cpuEnergy(c)$ = the energy usage of c on the CPU.
$cpuEnergy : C \rightarrow \mathbb{N}$	$cpuEnergy(c)$ = the energy usage of c on the GPU.

impose limitations over the system, as follows:

$$AvailCpuMem = \text{the available CPU memory}$$
$$AvailGpuMem = \text{the available GPU memory}$$
$$MaxEnergy = \text{ the maximum energy usage}$$
$$MaxTime_{\langle f_l, C_l \rangle} = \text{ the maximum execution time of a trigger set } C_\mathsf{l} \text{ with the frequency } f_\mathsf{l}$$

We define a *weight parameter* corresponding to each of the system property (see Section 5.2.3). They are used when constructing the optimization criteria (see second part of Section 5.2.4). The weight parameters are the following:

w^g = the optimization weight of the GPU memory usage function

w^c = the optimization weight of the CPU memory usage function

w^e = the optimization weight of the energy function

w^t_l = the optimization weight of each trigger set C_l of the system, where

$$w^\mathsf{g} + w^\mathsf{c} + w^\mathsf{e} + \sum_{\langle \mathsf{f}_l, \mathsf{C}_l \rangle \in \mathsf{S}} w^\mathsf{t}_\mathsf{l} = 1$$

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 is spent when transferring one unit of data from the CPU to GPU memory space, and vice-versa. Similarly, k_{tm} is the same when one unit of data is transferred from the CPU to GPU memory space, and vice-versa.

5.2.2 Output

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 constraints. 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 allocated to the GPU, in order to satisfy the optimization criteria. The allocation result is represented by a mapping between the components and the processing units (i.e., CPU/GPU). For example, using the application described in Figure 5.1, one of the possible allocation results that minimizes the CPU memory usage is described as follows:

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

Describing the result in a formal manner, let $A = \{a_{\mathsf{c}_1}, \ldots, a_{\mathsf{c}_n}\}$ 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.

$$a_{\mathsf{c}_i} = \begin{cases} 0, \text{ when } c_\mathsf{i} \text{ is allocated on CPU} \\ 1, \text{ when } c_\mathsf{i} \text{ is allocated on GPU} \end{cases}$$

5.2.3 System properties

We define four system properties, detailed in the following paragraphs.

$GpuMemory$

This property, presented by equation 5.1, describes the total amount of GPU memory required by the system components. $GpuMemory$ is composed of two parts, as follows. The first part 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} set to 1).

$$\begin{aligned} \mathsf{GpuMemory} = & \sum_{c_i \in C} \mathsf{a}_{c_i} * \mathsf{gpuMem}(\mathsf{c}_i) \quad + \\ & \sum_{op_i \in O} \mathsf{min}\Big(1, \mathsf{a}_{comp(op_i)} + \sum_{ip_j \in conn(op_i)} \mathsf{a}_{comp(ip_j)}\Big) * \mathsf{portSize}(\mathsf{op}_i) \end{aligned} \tag{5.1}$$

The second part 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.

These aspects are captured by the second part of the equation, which uses a *min* function to combine both aspects, as follows. For each output port op_{i} of the O set:

- if it belongs to a GPU-allocated component (i.e., $a_{\mathsf{comp}(\mathsf{op}_i)}$ 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., $a_{\mathsf{comp}(\mathsf{op}_i)}$ 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* operator 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 will be generated. 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.

CpuMemory

This property describes the total CPU memory usage of the system. It is calculated in a similar manner as the previous property, using two *sum* parts, as presented by equation 5.2.

$$\begin{aligned} CpuMemory = & \sum_{c_i \in C} \Big((1 - \mathsf{a}_{c_i}) * \mathsf{cpuMem}(\mathsf{c}_i) + \mathsf{a}_{c_i} * \mathsf{gpuCpuMem}(\mathsf{c}_i) \Big) \quad + \\ & \sum_{op_i \in O} \mathsf{min}\Big(1, 1 - \mathsf{a}_{comp(op_i)} + \sum_{ip_j \in conn(op_i)} (1 - \mathsf{a}_{comp(ip_j)})\Big) * \mathsf{portSize}(\mathsf{op}_i) \end{aligned} \tag{5.2}$$

The first part adds the internal CPU memory requirement of the components that are allocated on the CPU (i.e., components c_i with $(1 - a_{\mathsf{c}_i})$ as 1) and the CPU memory requirement of the components that are allocated on the GPU. The second part 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.

Energy

This property characterizes the total energy usage of the system. It 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 5.3. The energy usage of each of such component set C_l, described by equation 5.4, is obtained from two parts, as follows.

$$\mathsf{Energy} = \sum_{\langle f_l, C_l \rangle \in S} \mathsf{Energy}_{\langle f_l, C_l \rangle} * \mathsf{f}_l \tag{5.3}$$

$$\begin{aligned} \mathsf{Energy}_{\langle f_l, C_l \rangle} = & \sum_{c_i \in C_l} \Big(\mathsf{a}_{c_i} * \mathsf{gpuEnergy}(\mathsf{c}_i) + (1 - \mathsf{a}_{c_i}) * \mathsf{cpuEnergy}(\mathsf{c}_i) \Big) \quad + \\ & \sum_{op_i \in O \wedge comp(op_i) \in C_l} \Big(\mathsf{a}_{comp(op_i)} * \mathsf{min}\Big(1, \sum_{ip_j \in conn(op_i)} (1 - \mathsf{a}_{comp(ip_j)})\Big) \quad + \\ & (1 - \mathsf{a}_{comp(op_i)}) * \mathsf{min}\Big(1, \sum_{ip_j \in conn(op_i)} \mathsf{a}_{comp(ip_j)}\Big)\Big) * \mathsf{portSize}(\mathsf{op}_i) * \mathsf{k}_{eng} \end{aligned} \tag{5.4}$$

The first part calculates the energy usage of the components in the set C_l by adding each of the components' GPU energy usage, or CPU energy usage, depending on the allocation.

The second part 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 takes each output port op_i of the components c_i that are allocated on GPU (i.e., $a_{\mathsf{comp}(\mathsf{op}_i)}$ 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 energy used on transferring data from the CPU to GPU memory space is calculated. 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.

$Time_{\langle f_l, C_l \rangle}$

The property describes the end-to-end execution time for a set of components C_l activated by the same trigger source. The execution time of a particular set of components C_l is formally defined in equation 5.5, through the addition of two *sums*, described in the following paragraphs.

$$\begin{aligned}\mathsf{Time}_{\langle f_l, C_l \rangle} = & \sum_{c_i \in C_l} \Big(\mathsf{a}_{c_i} * \mathsf{gpuTime}(\mathsf{c}_i) + (1 - \mathsf{a}_{c_i}) * \mathsf{cpuTime}(\mathsf{c}_i)\Big) \;+ \\ & \sum_{op_i \in O \wedge comp(op_i) \in C_l} \Big(\mathsf{a}_{comp(op_i)} * \mathsf{min}\Big(1, \sum_{ip_j \in conn(op_i)} (1 - \mathsf{a}_{comp(ip_j)})\Big)\Big) \;+ \\ & \Big(\Big(1 - \mathsf{a}_{comp(op_i)}\Big) * \mathsf{min}\Big(1, \sum_{ip_j \in conn(op_i)} \mathsf{a}_{comp(ip_j)}\Big)\Big) * \mathsf{portSize}(\mathsf{op}_i) * \mathsf{k}_{tm}\end{aligned} \tag{5.5}$$

The first *sum* calculates the execution times of the system components allocated on the GPU and CPU, respectively. The second *sum* deals with the time spent on transferring data between the distinct memory spaces, similarly to the way energy is computed above. To compute the time spent on the transfer, the size of the output port (i.e., returned by the $portSize()$ function) is multiplied with the k_{tm} constant.

5.2.4 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 most 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, one constraint forces to place components on the GPU such that they together do not require more memory than is available. The optimization criteria are used to decide which of the feasible allocation solutions 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. In addition, 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.

The 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 (defined by equation 5.1) should not exceed the available GPU

memory of the platform:

$$GpuMemory \leq AvailGpuMem$$

2. CPU memory – similarly, the required memory of components allocated on the CPU (defined by equation 5.2) should not exceed the CPU's available memory:

$$CpuMemory \leq AvailCpuMem$$

3. Maximum energy – the energy consumed by the system (defined by equation 5.3) should be less or equal to a particular (maximum) limit:

$$Energy \leq MaxEnergy$$

4. Maximum time – the execution time of a particular trigger group (defined by equation 5.5) should be less or equal to a specific (maximum) execution time:

$$Time_{\langle f_l, C_l \rangle} \leq MaxTime_{\langle f_l, C_l \rangle}$$

5. All regular components must be allocated on the CPU:

$$a_{\mathsf{c}} = 0 \text{ for all } c \in R$$

The 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, we use the *weight parameters* introduced during the input part of our method (see Section 5.2.1). We merge all of the concerns in a linear combination, and minimize the result, as follows:

$$\begin{aligned}
&minimize(F), \text{ where} \\
&F = w^{\mathsf{g}} * GpuMemory + w^{\mathsf{c}} * CpuMemory + \\
&\quad w^{\mathsf{e}} * Energy + \sum_{\langle \mathsf{f}_l, \mathsf{c}_l \rangle \in \mathsf{S}} w^{\mathsf{t}}_{\mathsf{l}} * Time_{\langle \mathsf{f}_l, \mathsf{c}_l \rangle}, \text{ and}
\end{aligned}$$

5.2.5 Simplification of the system property equations

The system properties described in the previous section, are formalized in a general manner to cover systems with any number of components equipped with any number of ports. This leads to 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 5.2.4), 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 only has a single outgoing connection, to an input port of a component c_3, the first *min* function used for the energy calculation (i.e., equation 5.4), is initially $min(1, (1 - a_{\mathsf{c}_3}))$ but is reduced to just $(1 - a_{\mathsf{c}_3})$ that corresponds to the connected c_3 component. A more detailed example that describes the calculation and simplification of system properties, is presented in the next section.

5.3 Evaluation

The nine components of the vision system are divided in two trigger groups, i.e., group 1 containing six components (upper part of Fig. 5.2), and group 2 containing three components. 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.

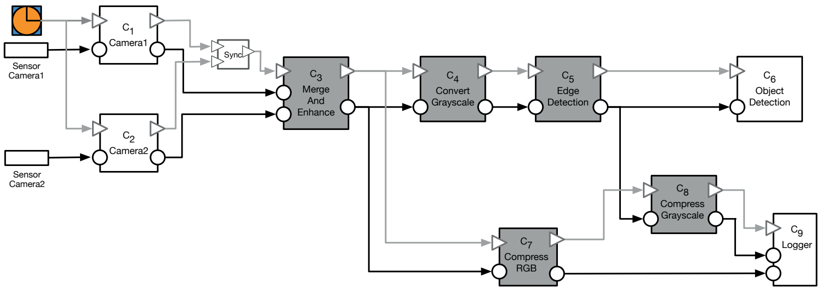


Figure 5.2: The Rubus design of the vision system

Using the described vision system, we conduct two experiments. While the first experiment examines the feasibility of the allocation method, the second experiment analyzes the optimization time of the solver.

5.3.1 Experiment 1

The system properties calculated using the equations from Section 5.2.3 have a simplified form. For instance, Table 5.2 presents the unrolled energy usage calculation for both vision system groups. Equation 5.6 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

Table 5.2: The unrolled and partly simplified energy equations for the two groups of the vision system

$$
\begin{aligned}
&Energy_{\langle \mathsf{f}_1,\mathsf{c}_1\rangle} = \\
&\quad cpuEnergy(c_1)+ \\
&\quad cpuEnergy(c_2)+ \\
&\quad a_{\mathsf{c}_3} * gpuEnergy(c_3) + (1 - a_{\mathsf{c}_3}) * cpuEnergy(c_3)+ \\
&\quad a_{\mathsf{c}_4} * gpuEnergy(c_4) + (1 - a_{\mathsf{c}_4}) * cpuEnergy(c_4)+ \\
&\quad a_{\mathsf{c}_5} * gpuEnergy(c_5) + (1 - a_{\mathsf{c}_5}) * cpuEnergy(c_5)+ \\
&\quad cpuEnergy(c_6)+ \\
&\quad a_{\mathsf{c}_3} * portSize(op_{\mathsf{c}_1}) * k_{\mathsf{eng}}+ \\
&\quad a_{\mathsf{c}_3} * portSize(op_{\mathsf{c}_2}) * k_{\mathsf{eng}}+ \\
&\quad (a_{\mathsf{c}_3} * min(1, 1 - a_{\mathsf{c}_4} + 1 - a_{\mathsf{c}_7}) + (1 - a_{\mathsf{c}_3}) * min(1, a_{\mathsf{c}_4} + a_{\mathsf{c}_7}))* \\
&\qquad portSize(op_{\mathsf{c}_3}) * k_{\mathsf{eng}}+ \\
&\quad (a_{\mathsf{c}_4} * (1 - a_{\mathsf{c}_5}) + (1 - a_{\mathsf{c}_4}) * a_{\mathsf{c}_5}) * portSize(op_{\mathsf{c}_4}) * k_{\mathsf{eng}}+ \\
&\quad (a_{\mathsf{c}_5} + (1 - a_{\mathsf{c}_5}) * a_{\mathsf{c}_8}) * portSize(op_5) * k_{\mathsf{eng}}
\end{aligned}
\tag{5.6}
$$

$$
\begin{aligned}
&Energy_{\langle \mathsf{f}_2,\mathsf{c}_2\rangle} = \\
&\quad a_{\mathsf{c}_7} * gpuEnergy(c_7) + (1 - a_{\mathsf{c}_7}) * cpuEnergy(c_7)+ \\
&\quad a_{\mathsf{c}_8} * gpuEnergy(c_8) + (1 - a_{\mathsf{c}_8}) * cpuEnergy(c_8)+ \\
&\quad cpuEnergy(c_9)+ \\
&\quad (a_{\mathsf{c}_7} * (1 - a_{\mathsf{c}_8}) + (1 - a_{\mathsf{c}_7}) * a_{\mathsf{c}_8}) * portSize(op_{\mathsf{c}_7}) * k_{\mathsf{eng}}+ \\
&\quad a_{\mathsf{c}_8} * portSize(op_{\mathsf{c}_8}) * k_{\mathsf{eng}}
\end{aligned}
\tag{5.7}
$$

(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 5.7. Furthermore, because c_9 (i.e., *Logger*) is a regular component (i.e., $a_{c_9} = 0$), this simplifies the equation even more.

To compute allocation schemes, we use the CPLEX solver[1] 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 5.3: The properties of the vision system components

Group	Component	Properties				
		Memory (bytes)			Time (msec)	
		CPU	GPU*	GPUCPU**	CPU	GPU
1	C1:Camera1	56	0	0	3.2	0
1	C2:Camera2	56	0	0	3.2	0
1	C3:MergeAndEnhance	69	10575	24	4	0.6
1	C4:ConvertGrayscale	60	8550	22	4	0.5
1	C5:EdgeDetection	132	24750	22	1	0.6
1	C6:ObjectDetection	124	0	0	6	0
2	C7:CompressRGB	100	15300	22	3.1	0.5
2	C8:CompressGrayscale	100	15300	22	3.2	0.5
2	C9: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 5.3. As we do not have means for energy usage measurements, we use literature results that show the GPU energy efficiency [40], and

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

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.

Regarding the platform, the embedded board of the underwater robot is equipped with an AMD Accelerated Processing Unit[2], 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 5.4: Four allocation scenarios for the vision system

Group	**Component**	**Allocation**			
		Scenario1	**Scenario2**	**Scenario3**	**Scenario4**
1	C3:MergeAndEnhance	GPU	CPU	CPU	GPU
1	C4:ConvertGrayscale	GPU	GPU	GPU	GPU
1	C5:EdgeDetection	CPU	GPU	GPU	CPU
2	C7:CompressRGB	GPU	GPU	GPU	GPU
2	C8:CompressGrayscale	CPU	GPU	GPU	GPU

We considered four scenarios when computing the allocation schemes:

- *Scenario1* minimizes the execution time of group 1 (i.e., $w_1^{\mathsf{t}} = 1$, and all other weights set to 0).
- *Scenario2* minimizes the execution time of group 2 (i.e., $w_2^{\mathsf{t}} = 1$, and all other weights set to 0).

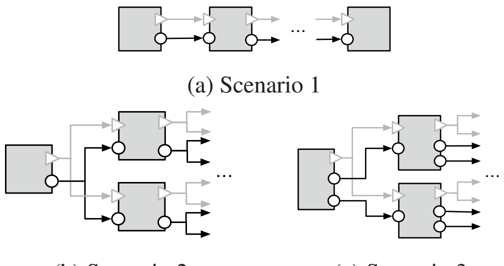
[2]https://unibap.com/product/advanced-hetereogeneous-computing-modules/

- *Scenario3* minimizes all system properties, where all weights are equal and set to 0.2.
- *Scenario4* minimizes all system properties, where the CPU memory weight parameter w^{c} is double than all other weights).

Table 5.4 presents the four resulting allocation schemes of the vision system computed by the CPLEX solver when using a machine with a 2,6GHz i7 CPU and 16 GB of memory. For the *Scenario1* case, 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, probably due to the memory limitation constraints. In the second scenario, 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 *Scenario3* case, all flexible components, except *MergeAndEnhance*, are distributed over the GPU. In the last case, all flexible components except *EdgeDetection*, are allocated on the GPU.

5.3.2 Experiment 2

For the vision system, the time used by the solver to compute the allocation schemes is under 100 milliseconds per scenario. To examine the calculation time for bigger systems, we constructed three scenarios described in Fig. 5.3.



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

Figure 5.3: Three types of scenarios for scalability evaluation

Scenario 1 consists of a chain of connected flexible components. In Scenario 2, each flexible component communicates, via a single output port, with two other flexible components, and in Scenario 3, each flexible component has two output ports connected to different components. In each scenario we then vary the total number of components. An odd number of components is needed for the connections in Scenario 2 and 3. We decided to use the same odd number of components in all three scenarios, resulting in three versions with 21, 31 and 41 components, respectively.

Table 5.5: Implementation and execution of the scenarios

	Number of				Average
	flexible components	output data ports	data connections	total operators	optimization time (msec)
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	6040
Scenario 3	21	38	38	1089	174
	31	58	58	1661	520
	41	78	78	2235	21374

Each scenario version was implemented in CPLEX by unrolling the general equations (i.e., equation 5.1, equation 5.2, equation 5.3 and equation 5.5). Table 5.5 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. While the available CPU memory property is set to a value higher than the CPU usage of all components, the available GPU memory property is set to half of the GPU usage of all components. The optimization was ran 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 i7 CPU.

Fig. 5.4 gives a more detailed perspective of the time spent computing solutions for the considered scenarios. We notice that the solver easily handles all the considered scenarios, in less than 100 seconds. We mention that in all simulations, either an optimal solution was found or no solution existed. The experiment shows that the optimization time is mainly influenced by the number of flexible components, but factors such as the number of output data ports and their distinct connections also seem to impact the optimization time.

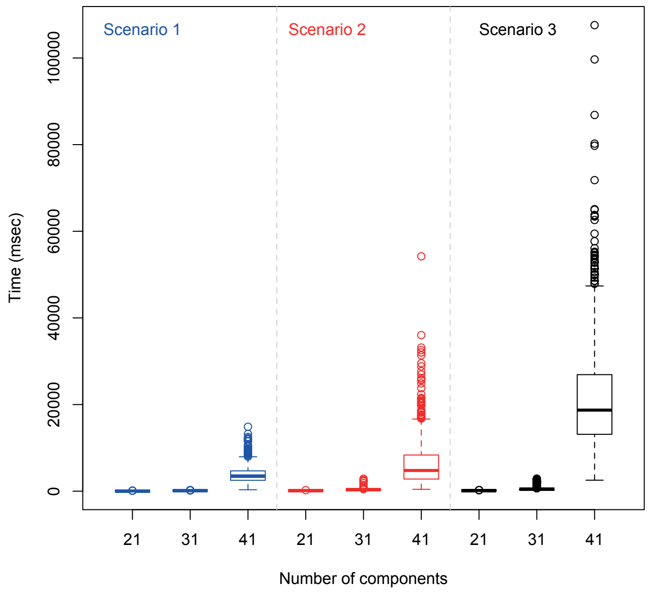


Figure 5.4: The execution time used for computing solutions

This experiment shows that our allocation method is feasible even for fairly complex systems. In particular, when the number of forked data connections is low, the method scales well.

VI

Chapter 6

Related work

This chapter presents contributions divided into three parts. The first part introduces the support for designing systems with GPUs, the second part presents used programming models to address the GPU, and the third part covers the allocation optimization contributions. These parts include works that are relevant to discuss in the context of the novelties introduced in this thesis.

6.1 Support for heterogeneous system design

The latest technological progress has facilitated the development of Systems-on-Chip (SoC) with multiple heterogeneous processors (e.g., CPU, GPU, FPGA) into a single chip. In this sense, Andrews et al. propose the usage of COTS components to address SoC systems with CPUs and FPGAs [41]. The authors developed, based on the multithreading POSIX programming paradigm, an interface abstraction layer to ease the component synchronization over the shared memory. The work targets embedded applications composed of components that concurrently execute and synchronize and exchange data. Although the work focuses on creating a model for CPU-FPGA systems, the authors point out that the existing way to develop applications, i.e., developing components specific to the CPU/FPGA, is opposite to the desired goals (e.g., modularity, portability, reuse). In the context of component development of systems with GPU capability, we introduce the work of Bernier et al. [42]. The authors present a way of developing, using the OpenCL framework, military radio applications that use platforms with GPUs. The work describes how components encapsulate the data transfer activities, which makes them platform specific and greatly affecting the reusability aspect.

Regarding systems with GPU capabilities, we mention a general-purpose component model called PEPPHER [38] that proposes a way to efficiently utilize CPU-GPU hardware. In this sense, the authors define the PEPPHER component as an annotated software unit. The component interface is described by an XML document that contains the name, parameter types and access type of the component. Dealing with platforms with different processing units (i.e., CPUs and GPUs), the interface may define several implementation variants of the same component. Regarding the data passed between PEPPHER components, this is wrapped into portable and generic data structures called *smart containers*. These structures ensure the data transfer between the memories of the processing units. In our work, we provide a transparent and automatic way of transferring data between processing units by using adapters. Similar to the smart container concept which is more complex when dealing with memory management, our adapters can be considered as high-level memory management elements. Moreover, regarding the different PEPPHER component variants, our work's advantage is that it has less overhead (e.g., memory footprint, development time) by using a single flexible component. After the component is allocated to either the CPU or GPU, automatic means generate the appropriate variant of the flexible component.

We also mention the Global Composition Framework (GCF) that extends the PEPPHER component model [43]. Although GCF focuses on performance optimization, the work addresses the development of systems with GPUs. The considered component model defines the notion of component that consists of an interface. The interface describes a functionality and the functionality is implemented by multiple variants. The interfaces and implementation variants have attached meta-data (e.g., to target GPU). While in PEPPHER, the meta-data is described by an XML, under GCF, pragmas are used to represent the meta-information. The data used by a component needs to reside in the memory space associated with the component's execution platform. The framework takes care of the data transfer between components that reside in disjoint memory spaces, using runtime support. This is done via a static data analyzer that checks the program data flow in order to find out where to place data handling code. The runtime support allows the data transfer only when an implementation gets called, minimizing the memory overhead. However, it is difficult to use runtime assistance in the domain targeted in our work given that a transfer that is unknown when is realized, may present a big overhead for a control-type of system.

Another work that targets heterogeneous systems with e.g., CPU, GPU and FPGA is the Elastic computing framework [44]. Although is not a component model per se, this framework has similar principles (e.g., interfaces, usage of already developed functions). The framework uses a library that contains the so called *elastic functions*. An elastic function has different implementations. For example, an implementation may use the CPU and GPU, while another uses only the CPU. The framework, being focused on the system performance, analyzes the execution time of the elastic functions for all combinations of resources, and decides during run-time, the fastest implementation for a given combination of resources. The Elastic framework handles resource allocation and data management inside the elastic function. An improvement provided by our work is that it externalizes the data management outside the component (i.e., using adapters), in this way enhancing the component reusability.

Brock et al. extend an existing library (i.e., PVTOL) in order to support GPU architecture [45]. The authors take in consideration multiple memory architectures which are handled through *conduits*. Basically, a conduit is a template, where, based on its arguments, it targets the desired platform (i.e., CPU or GPU). The extended library has similarities with our work. While, in Brock's work, the developer introduces, at the source-code level, the conduits to address the desired platform, in our work we leave this choice to the designer, that, after the application architecture is constructed, decides which platform to be use for the flexible components. Another aspect targeted by Brock et al. is the memory management which is also done via conduits, implemented by the developer. In our work, (part of) the memory management is done automatically via the adapter artifacts.

Another work that uses a library to address the GPU capabilities in mathematics, is introduced by Winter et al. [46]. The library converts data-parallel expressions into kernel functions written using the PTX programming language (i.e., an assembler language which is intermediary between high-level CUDA C/C++ code and GPU machine code). Furthermore, the library automates the memory access via a *cache* and *page-out* mechanisms. The cache mechanism extracts required data fields and make them available to the GPU, which are paged-out (i.e., copied to the CPU) when they are accessed by the CPU or there is not enough memory to hold other cached data fields. Due to the narrow focus (i.e., mathematics) of this library, the mechanism that automatically generates kernel functions cannot apply to our work. Furthermore, the used PTX language increases the demands on the developer to posses knowledge in (low-level) GPU development. Regarding the memory management, the *page-out* mechanisms cannot be utilized in our work given the introduced unpredictability of data transfer activities.

Other works use a different approach, i.e., model-driven engineering, for development of SoC embedded systems. For example, Gamatie et al. [47] present the GASPARD design framework for massively parallel embedded systems. Designing the systems at a high abstraction level using the MARTE standard profile, the GASPARD framework allows the designers to automatically generate code of high-performance embedded systems. Another work that is worth mentioning is the of work of Rodrigues et al. [48]. The authors extend the MARTE profile to allow modeling of GPU architectures. This work, as well as the GASPARD framework, introduce mechanisms to handle the GPU memory system and its interaction with the main memory system. Being developed in 2011, both frameworks focus on systems only with distinct (CPU and GPU) memory systems. In our work, we cover newer platforms with GPUs that have different characteristics regarding the memory access. The components proposed in our work, complemented by the adapter mechanisms, allow the transparent usage of many of the existing embedded platforms with GPUs.

Flexibility for embedded systems is addressed in different ways and for different reasons. For example, Lednicki et al. [49] 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 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.

A way to better manage the complexity of component-based systems is to use composite components. A composite component, composed of several flat components, follows the same rules (e.g., encapsulation) as regular components, and its functionality is given by the combination of enclosed components. To support analysis techniques for component-based systems, Lévêque et al. introduce a solution to flatten systems with composite components [50]. Accordingly, the solution breaks a composite component into its constituent components, and realizes the (data and control) connections with the rest of the system. Other work that splits a composite component, is used in distributed systems, where the decomposed components are executed on different execution nodes [51]. One of our introduced solution acts in the opposite direction in order to reduce the system overhead. From a flat component-based system, we enclose connected components in groups and “compress” the groups in conceptual components.

The functionality of a flexible group is composed from the functionalities of the enclosed components (i.e., kernel functions). We execute the kernel functions in the order that the components are enclosed in the flexible group set. The performance of a system may be improved through the composition of the kernels [52]. Sarkar et al. introduce a methodology to compose kernels such that the resulting system performance is improved over the sum of the performances of the individual kernels.

Flexibility in component-based development is highly desirable in order to break the barrier of domain-specific components. In this sense, a developed design approach advertises the usage of common component platform with various plug-ins [53]. The advantage of the approach is that the component platform can be re-purposed to meet new domains by using new plug-ins, instead of re-writing the original component platform. Similarly, the “flexible” notion that we extend from the component to group concept, increases the designer options when constructing a component-based system which may lead to e.g., a more efficient system.

6.2 Programming models and code generation for GPU usage

There exist different programming models that target GPUs. Figure 6.1 presents few representative solutions that facilitate the GPU development. Two dimensions are presented in the figure, where the horizontal axis covers the development manner (i.e., manual and automatic), and the vertical axis presents the management of the CPU-GPU communication.

CUDA [28] is a framework developed by NVIDIA, targeting only NVIDIA GPUs. Brook [54], a framework used to develop general-purpose graphics hardware, addresses AMD and NVIDIA GPUs. OpenCL [29] is another framework that bridges the gap between CPU and GPU, being employed even further to address other processing unit types such as FPGA. For the three previously mentioned frameworks, the communication between processing units falls under the developers responsibility, being a manual activity.

Other frameworks that address the GPU development in an automatic way are CUDA-lite [55], a C-to-CUDA code generator [56], and the R-Stream compiler [57]. For example, CUDA-lite translates CUDA functions into functions with improved performance by coalescing and exploiting the GPU memory. The downside of these works is that the developer needs to manually address the data transfer between the CPU and GPU.

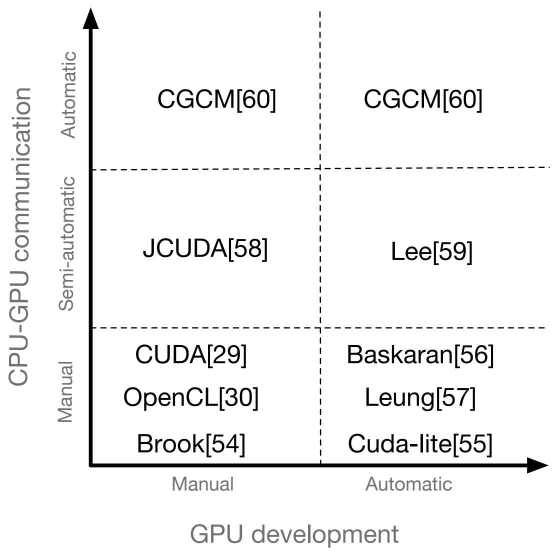


Figure 6.1: Taxonomy of related work on GPU development

Moving one level up on with respect to the communication management, we mention the JCUDA framework [58] that uses Java and the framework proposed by Lee et al. [59]. Both frameworks handle in a semi-automatic way the CPU-GPU communication. For example, JCUDA delegates the generation of the host-device data transfer activities to the compiler. On the highest level of the vertical axis, there is the CGCM system that, in a fully automatize manner, manages and optimizes the CPU-GPU communication [60].

Given that OpenCL is well established in industry, being supported by most of the existing vendors (e.g., Intel, AMD, NVIDIA, Xilinx), the usage of OpenCL to develop our introduced solutions, increases the feasibility of this thesis in the current industrial context. CUDA (or its extensions, e.g., CUDA-lite) was also a feasible option, being a popular and known development framework for GPUs. The downside resides on its addressed platforms, i.e., only NVIDIA GPUs. The other presented frameworks (e.g., CGCM, R-Stream) are fairly new developed and not used in industry.

Regarding the GPU code generation, Diogo et al. [61] present a compiler and runtime systems extensions for Single Assignment C (SaC), a functional array programming language. Using the introduced support, the developer does not need to explicitly write source-code for platforms with GPUs; the solution ports, on the GPU, the loop instructions that are capable to be executed in parallel. Furthermore, the memory management is facilitated via dedicated conversion functions. These functions performs the copying of only the needed data blocks, instead of the entire array. As opposite, the adapter artifacts proposed in our work, copy the entire data. Given the fact that the component encapsulates all its information, and not having additional information on how the data is used inside the components, the adapters cannot transfer only parts of data. Furthermore, in our work, the functionalities of the flexible components can be very complex and impossible to be automatically generated as opposite to the simple loop instructions used in Diogo's work.

To improve the programmability for mainstream programmers, Sandrieser et al. [62] introduce a framework for heterogeneous multi-core systems. The approach differs between mainstream and expert programmers, where the mainstream developer constructs the application at a higher abstraction level using software artifacts created by an expert programmer that have in-depth knowledge on platform-specific development. The functions (i.e., software artifacts) of the application that are to be executed on a specific platform (e.g., GPU), are annotated by the mainstream programmer with source code directives that target the desired platform, while the expert programmer provides different implementation variants for various platforms. The compiler, using platform description meta-data, automatically selects the appropriate variant corresponding to the required platform. In our work, we also distinguish between two types of developers, i.e., the component and the system developers that correspond to the expert and mainstream programmers, respectively. Through our introduced flexible component concept, we ease the work of the component developer, which constructs only one version of the component functionality. This functionality (encapsulated inside a component) is ported on the platform selected by the system developer.

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. [63] 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.

We also mention several works that construct APIs to decrease the programming complexity of GPUs. Fu et al. introduces a simple API that, encapsulating the complexity of GPU architectures, allows transparent optimization strategies for implementation of different graph algorithms [64]. The API, constructed on top of the CUDA model, is limited to only target NVIDIA GPUs. Another API that provides function calls to run common vision algorithms, is OpenVIDIA [65]. The functions of the API abstract OpenGL calls needed for vision processing. An improved framework, i.e., MapCG [66], contains a high level API (C-like language) that hides programming side-burdens (e.g., communication). Similarly, we include a generic API (see Section 4.6.1) that provides platform-independent functions. In our work, the API consists of only four functions, which makes it small with a reduced memory overhead impact. From the component communication perspective, there is no need for an API in our work as transparent communication mechanism (i.e., adapters) are automatically generated.

6.3 Allocation optimization

The fourth research goal deals with the software-to-hardware allocation. There is a lot of existing work that addresses this issue, such as the systematic literature review on the architecture optimization methods [67]. 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 [68]. 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. [69]. 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 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 [70]. 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.

We introduced a static allocation method, where an allocation result is computed a priori, and used once the application is executed. A different approach would be to compute allocation schemes during run-time. In the same embedded system context, we mention an interesting work of Singh et al.[71] which performs energy efficient allocations of concurrent applications on CPU-GPU cores of heterogeneous MPSoC. This work handles the application at a lower granularity level (i.e., thread-level) than our work, and decides, during run-time, where to place (i.e., either on the CPU or GPU) the working threads in order to maximize the energy efficiency. Furthermore, this work does not cover, as our proposed allocation method, the energy consumed on data transfer between CPU and GPU memory systems, being considered as a future work direction.

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 [72]. 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 [73]. 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 [40]. 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 [74].

VII

Chapter 7

Conclusions and future work

7.1 Summary and conclusions

Component-based development offers no specific support for the construction of embedded systems with GPUs. In this context, the existing approaches introduce several shortcomings which diminish the benefits of using CBD in embedded systems with GPUs. The main goal of this thesis is to facilitate the component-based development of embedded systems with GPUs. More concretely, we defined three (sub-)goals to be addressed in this thesis. The first goal aims to identify the GPU functionality support provided by existing component-based frameworks. Based on the findings of the first goal, we intend to assist the development of embedded systems with GPUs via specific mechanisms. The last goal deals with the software-to-hardware allocation, were our target is to handle the allocation of the functionality onto the physical platform.

The contribution of the thesis addresses the overall and the three specific goals of the thesis, and are presented in the following paragraphs:

1. The first contribution presents how the existing solutions handle the GPU support. Using a systematic literature review methodology, we studied the state-of-the-art of component-based applications with GPU capabil-

ity and identified their trends and research specific details, briefly presented in next paragraph.

The interest in the development of component-based systems with GPUs started from 2009, a possible reason being the release on the market of several (software and hardware) GPU technologies. Moreover, most of the conducted research is done in academia. Regarding the specific details, most of the captured research use no specific component model, and the two most used mechanisms to develop solutions are via programming and modeling.

2. The second contribution introduced specific GPU mechanisms as follows. A *flexible component* is introduced, which can be executed with no modifications required, on either CPU or a variety of GPU platforms. The required information to allow the component to be executed on the selected platform is automatically generated. An optimization step is also introduced, were connected flexible components that share the same executable platform (i.e., CPU or GPU) are grouped in entities that behave as regular components. To facilitate the communication between components, we introduce artifacts called *adapters* to handle data transfer activities between connected components. The adapters are automatically generated when needed, depending on the platform characteristics and on which executable platform components are placed.

3. The theoretical concepts introduced by the previous contribution are implemented as an extension of the Rubus component model, as follows. The flexible components were realized as regular Rubus components, characterized by interfaces, constructors, behavior functions and destructors. Similarly, we realized the connected flexible components into single regular Rubus components, following the optimization step via the flexible group concept. Adapters were also realized as regular Rubus components.

4. The last contribution of the thesis is a method for allocation optimization of component-based applications over platforms with GPU hardware.

Having components with flexible functionality (i.e., can be executed either by the CPU or GPU), we introduce a method to provide allocation schemes of the system's components, with respect to various optimization criteria (e.g., system performance).

Through the introduced novelties, we facilitate the development of embedded systems with GPUs and provide the following benefits:

1. We provided a reusable SLR framework that can serve as a starting point for new literature review studies. Furthermore, an up-to-date review of the state-of-the-art can be used by researchers that are interested in the research domain.
2. The introduced specific GPU mechanisms bring the following advantages:
    - Introducing a type of component that can be executed on different processing units, without manual modification, increases the *flexibility* of designing embedded systems.
    - The flexible component is platform-independent, being capable to be executed on platforms with GPUs that have different characteristics. For example, the same component can be executed, with no modification required, on platforms that have distinct CPU and GPU memory systems but also on platforms that fully share the same memory. This increases the component *reusability*.
    - Due to the mechanisms that automatically generate the required information which the component requires to be successfully executed on selected platform, the development of components with GPU capability is simplified. The adapter is another factor that decreases the development effort due to the fact that the developer is not responsible anymore to handle specific data transfer activities.
    - The component *maintainability* is also increased by the adapters. Through this concept, the specific data transfer activities are not encapsulated inside the component anymore, but automatically handled.

3. Through the introduced allocation method, we provide optimized allocation schemes in a semi-automatic way. The method takes as input the system characteristics, and compute optimized component-to-hardware schemes (when existing) with respect to different system properties (e.g., memory and energy usage).

The introduced GPU mechanisms were implemented by extending a state-of-practice component model (i.e., Rubus). The extended component model was used to construct a vision system and, through it, evaluated the feasibility of the proposed mechanisms. Similarly, the allocation method was implemented using a mathematical solver and was used to analyze the produced allocation results. Through the conducted evaluation, we argued the feasibility of the introduced novelties that facilitate component-based development of embedded systems with GPUs.

7.2 Future work

Several future work directions remain to continue our work:

- As there are various types of heterogeneous embedded systems such as Spartan 6 equipped with FPGA[1], ways to apply and/or extend our work on these types of systems can be considered. FPGA is a processing unit that excels in parallel processing data, being an alternative to the GPU. Due to the fact that an FPGA is equipped with a private memory system, the adapters artifacts may be adopted to facilitate the communication between components allocated on CPU and FPGA. Furthermore, given that the OpenCL framework provides also support for this type of systems, the implementation of the adapters artifacts may be directly use for this type of embedded systems.

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

- As the GPU is a parallelizible processing unit, a future work direction includes the parallel scheduling of components with GPU capability. For example, if the hardware has much resources, several components with GPU capability may be run in parallel. This future work direction may complemented the previous one, where multi-cores simultaneous access the GPU and execute, in parallel, their functionality.

- Within the same parallelism direction, and related to the grouping of flexible components with the same allocation into single entities, another approach to extend this thesis is to introduce mechanisms to allow

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

the parallel execution of the grouped functionalities. The mechanisms should analyze which functionalities are independent of each other and permit their parallel execution, according to the platform resources.

- The allocation model introduced in the fourth contribution, may be extended to address all types of platforms considered in this work. Besides the platform that has CPU and GPU distinct memory systems, platform with e.g., full shared memory and virtual shared memory should be covered by our method. Furthermore, more system properties (e.g., CPU and GPU utilization) and optimization criteria (e.g., end-to-end execution time) may be introduced in order to provide more exact solutions.

Bibliography

[1] Global Market Insights. Embedded system market size by application, by product. Industry outlook report, regional analysis, application development potential, price trends, competitive market share and forecast, 2016 - 2023, 2006.

[2] Alfred Helmerich, Nora Koch, Luis Mandel, Peter Braun, Peter Dornbusch, Alexander Gruler, Patrick Keil, Roland Leisibach, Jan Romberg, Bernhard Schätz, et al. Study of worldwide trends and R&D programmes in embedded systems in view of maximising the impact of a technology platform in the area. *Final Report for the European Comission, Brussels, Belgium*, 2005.

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

[4] AUTOSAR - Technical Overview. http://www.autosar.org. Accessed: 2018-07-27.

[5] Karl Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.

[6] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus component model for

resource constrained real-time systems. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 177–183. IEEE, 2008.

[7] Victor R Basili. The experimental paradigm in software engineering. In *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 1–12. Springer, 1993.

[8] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and software technology*, 55(12):2049–2075, 2013.

[9] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools*. "O'Reilly Media, Inc.", 2006.

[10] Augusto Vega, Pradip Bose, and Alper Buyuktosunoglu. *Rugged Embedded Systems: Computing in Harsh Environments*. Morgan Kaufmann, 2016.

[11] Vijaykrishnan Narayanan and Yuan Xie. Reliability concerns in embedded system designs. *Computer*, 39(1):118–120, 2006.

[12] Marilyn Wolf. *High-performance embedded computing: applications in cyber-physical systems and mobile computing*. Newnes, 2014.

[13] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE software*, 26(3):19–25, 2009.

[14] Szyperski Clemens, Gruntz Dominik, and Murer Stephan. Component software: beyond object-oriented programming, 1998.

[15] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron. A classification framework for software component models. *IEEE Transaction of Software Engineering*, 37(5):593–615, October 2011.

[16] Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[17] OMG CORBA Component Model. http://www.omg.org/spec/CCM/4.0/. Accessed: 2018-07-27.

[18] Juval Lowy. *Programming. NET Components: Design and Build. NET Applications Using Component-Oriented Programming*. " O'Reilly Media, Inc.", 2005.

[19] Sun Microsystems. JavaBeans Specifications. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html/. Accessed: 2018-07-27.

[20] Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. ProCom-the Progress Component Model Reference Manual. *Malardalen University, Vasteras, Sweden*, 2008.

[21] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 199–208. IEEE, 2007.

[22] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.

[23] Sara Dersten, Jakob Axelsson, and Joakim Froberg. An analysis of a layered system architecture for autonomous construction vehicles. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*, pages 582–588. IEEE, 2015.

[24] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96, 2008.

[25] Svetlin A Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2007.

[26] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, 228(12):4468 – 4477, 2009.

[27] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 2007.

[28] NVIDIA CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2018-07-27.

[29] The OpenCL specification version 2.1. https://www.khronos.org/developers/reference-cards/. Accessed: 2018-07-27.

[30] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[31] David Budgen and Pearl Brereton. Performing systematic literature reviews in software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 1051–1052. ACM, 2006.

[32] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *EASE*, volume 8, pages 68–77, 2008.

[33] Trisha Greenhalgh and Richard Peacock. Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources. *Bmj*, 331(7524):1064–1065, 2005.

[34] Daniela S Cruzes and Tore Dybå. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455, 2011.

[35] Mark Rodgers, Amanda Sowden, Mark Petticrew, Lisa Arai, Helen Roberts, Nicky Britten, and Jennie Popay. Testing methodological guidance on the conduct of narrative synthesis in systematic reviews: effectiveness of interventions to promote smoke alarm ownership and function. *Evaluation*, 15(1):49–73, 2009.

[36] Jacob Cohen. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin*, 70(4):213, 1968.

[37] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[38] Usman Dastgeer, Lu Li, and Christoph Kessler. The PEPPHER composition tool: Performance-aware dynamic composition of applications for GPU-based systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 711–720. IEEE, 2012.

[39] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly, and Tamara Dahlgren. The CCA component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience*, 18(2):215–229, 2006.

[40] Song Huang, Shucai Xiao, and Wu Feng. On the energy efficiency of graphics processing units for scientific computing. In *Parallel & Distributed Processing, IEEE International Symposium on*. IEEE, 2009.

[41] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp, and Peter Ashenden. Pro-

gramming models for hybrid FPGA-CPU computational components: a missing link. *IEEE Micro*, 24(4):42–53, 2004.

[42] Steve Bernier, François Lévesque, Martin Phisel, Dmitry Zvernik, and David Hagood. Using OpenCL to increase SCA application portability. *Journal of Signal Processing Systems*, 89(1):107–117, 2017.

[43] Usman Dastgeer and Christoph Kessler. A framework for performance-aware composition of applications for GPU-based systems. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 698–707. IEEE, 2013.

[44] John R Wernsing and Greg Stitt. Elastic computing: A portable optimization framework for hybrid computers. *Parallel Computing*, 38(8):438–464, 2012.

[45] James Brock, Miriam Leeser, and Mark Niedre. Heterogeneous tasks and conduits framework for rapid application portability and deployment. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*, pages 1–9. IEEE, 2012.

[46] FT Winter, Mike A Clark, Robert G Edwards, and Bálint Joó. A framework for lattice QCD calculations on GPUs. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1073–1082. IEEE, 2014.

[47] Abdoulaye Gamatie, Sebastien Le Beux, Eric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(4):39, 2011.

[48] Antonio Wendell De Oliveira Rodrigues, Frederic Guyomarc'h, and Jean-Luc Dekeyser. A modeling approach based on UML/MARTE for GPU architecture. *arXiv preprint arXiv:1105.4424*, 2011.

[49] Luka Lednicki, Juraj Feljan, Jan Carlson, and Mario Zagar. Adding support for hardware devices to component models for embedded systems. In *The Sixth International Conference on Software Engineering Advances*, 2011.

[50] Thomas Lévêque, Jan Carlson, Séverine Sentilles, and Etienne Borde. Flexible semantic-preserving flattening of hierarchical component models. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 31–38. IEEE, 2011.

[51] Ansgar Radermacher, Önder Gürcan, Arnaud Cuccuru, Sébastien Gérard, and Brahim Hamid. Split of composite components for distributed applications. In *Languages, Design Methods, and Tools for Electronic System Design*, pages 265–280. Springer, 2015.

[52] Santonu Sarkar, Sayantan Mitra, and Ashok Srinivasan. Reuse and refactoring of GPU kernels to design complex applications. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 134–141. IEEE, 2012.

[53] Marcus A Rothenberger, Hemant Jain, and Vijayan Sugumaran. A platform-based design approach for flexible software components. *Journal of Information Technology Theory and Application (JITTA)*, 18(2):3, 2017.

[54] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*. ACM, 2004.

[55] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wenmei. CUDA-lite: Reducing GPU programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 2008.

[56] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[57] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010.

[58] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par 2009 Parallel Processing*, pages 887–899. Springer, 2009.

[59] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[60] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic CPU-GPU communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151. ACM, 2011.

[61] Miguel Diogo and Clemens Grelck. Towards heterogeneous computing without heterogeneous programming. In *International Symposium on Trends in Functional Programming*, pages 279–294. Springer, 2012.

[62] Martin Sandrieser, Siegfried Benkner, and Sabri Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 17–24. ACM, 2011.

[63] Mingliang Wang and Manish Parashar. Object-oriented stream programming using aspects. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[64] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[65] James Fung and Steve Mann. OpenVIDIA: Parallel GPU computer vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, pages 849–852. ACM, 2005.

[66] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010.

[67] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 2013.

[68] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 421–424. ACM, 2007.

[69] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos, Slobodan Matic, Bodhi Priyantha, and Feng Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *Proceedings of the 45th annual design automation conference*, pages 191–196. ACM, 2008.

[70] Shige Wang, Jeffrey R Merrick, and Kang 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.

[71] Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V Merrett, and Bashir M Al-Hashimi. Energy-efficient run-time mapping

and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):147, 2017.

[72] Gabriel Campeanu, Jan Carlson, and Séverine Sentilles. Component allocation optimization for heterogeneous CPU-GPU embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 229–236. IEEE, 2014.

[73] Gabriel Campeanu and Mehrdad Saadatmand. Run-time component allocation in CPU-GPU embedded systems. In *Proceedings of the Symposium on Applied Computing*, pages 1259–1265. ACM, 2017.

[74] Martin Burtscher, Ivan Zecena, and Ziliang Zong. Measuring GPU power with the K20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 28. ACM, 2014.

One pressing challenge of many modern embedded systems is to successfully deal with the considerable amount of data that originates from the interaction with the environment. A recent solution comes from the use of GPUs. Equipped with a parallel execution model, the GPU excels in parallel processing applications, providing an improved performance compared to the CPU.

Another trend in the embedded systems domain is the use of component-based development. This software engineering paradigm that promotes construction of applications through the composition of software components, has been successfully used in the development of embedded systems. However, the existing approaches provide no specific support to develop embedded systems with GPUs. As a result, components with GPU capability need to encapsulate all the required GPU information in order to be successfully executed by the GPU. This leads to component specialization to specific platforms, hence drastically impeding component reusability.

Our main goal is to facilitate component-based development of embedded systems with GPUs. We introduce the concept of flexible component, which increases the flexibility to design embedded systems with GPUs, by allowing the system developer to decide where to place the component, i.e., either on the CPU or GPU. Furthermore, we provide means to automatically generate the required information for flexible components corresponding to their hardware placement, and to improve component communication. Through the introduced support, components with GPU capability are platform-independent, being capable to be executed on a large variety of hardware (i.e., platforms with different GPU characteristics). Furthermore, an optimization step is introduced, which groups connected flexible components into single entities that behave as regular components. Dealing with components that can be executed either by the CPU or GPU, we also introduce an allocation optimization method. The proposed solution, implemented using a mathematical solver, offers alternative options in optimizing particular system goals (e.g., minimize memory and energy usage).

MÄLARDALEN UNIVERSITY
SWEDEN

Address: P.O. Box 883, SE-721 23 Västerås. Sweden
Address: P.O. Box 325, SE-631 05 Eskilstuna. Sweden
E-mail: info@mdh.se **Web:** www.mdh.se

ISBN 978-91-7485-393-3
ISSN 1651-4238