Parallel Execution Optimization of GPU-aware Components in Embedded Systems

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

***Abstract*—Many embedded systems process huge amount of data that comes from the interaction with the environment. The Graphics Processing Unit (GPU) is a modern embedded solution that tackles the efficiency challenge when processing a lot of data. GPU may improve even more the system performance by allowing multiple activities to be executed in a parallel manner. In a complex component-based application, the challenge is to decide the components to be executed in parallel on GPU when considering different system factors (e.g., GPU memory, GPU computation power).**

In the context of component-based CPU-GPU embedded systems, we propose an automatic method that provides parallel execution schemes of components with GPU capabilities. The introduced method considers hardware (e.g., available GPU memory), software properties (e.g., required GPU memory) and communication pattern. Moreover, the method optimizes the overall system performance based on component execution times and system architecture (i.e., communication pattern). The validation uses an underwater robot example to describe the feasibility of our proposed method.

***Keywords*—CBD, component-based development, CPU-GPU, embedded systems, GPU-aware component, GPU component, parallel component execution, optimization**

I. Introduction

An embedded system is a computational systems that may be part of a larger system executing one or several specific tasks. Many of the modern embedded systems deal with huge amount of data due to e.g., interaction with the environment. For example, underwater robots use their sensors (e.g., cameras, radars) to receive information about the surrounding underwater environment and process it in order to e.g. detect various objects.

Due to the limited computing power and the sequential execution model of the CPU, traditional embedded systems have a challenge in providing the required performance level demanded by applications, while processing huge amount of data. For example, an underwater robot, while navigates, needs to identify with a certain performance the encountered objects in order to cope with the environment changes such as detecting moving objects.

One feasible solution that tackles the efficiency challenge of processing huge amount of data is the GPU. Due to its parallel execution architecture, the GPU can efficiently process data in a parallel manner. Today, using the latest technological improvements, there are embedded boards that contain GPUs also know as CPU-GPU embedded boards such as NVIDIA Jetson TK1 and AMD Kabini.

Another trend in the development of embedded systems is the usage of component-based development (CBD) [1] [2]. This software engineering methodology promotes development of systems through the composition of already existing software units known as (software) components. A key concept of CBD is the encapsulation, where components are seen as black boxes and the source code is encapsulated (inside the components). CBD is an attractive solution for embedded systems due to e.g., its faster time-to-market and the increase of the development productivity. The industry has successfully adopted CBD for embedded system development through component models such as AUTOSAR, Koala, Rubus and IEC 61131.

In a component-based CPU-GPU embedded system, the GPU, besides an increased efficiency w.r.t. data processing, may improve the system performance by allowing multiple components to be processed in parallel. Due to the fact that the GPU is characterized by a different architecture then the traditional CPU, different rational needs to be used when targeting GPU parallelism. The challenge of specifying how many components may be parallel executed relies not only on the available hardware resource description (e.g., available GPU memory) but also on the component specifications (e.g., GPU memory usage) and the system communication pattern. For example, the sum of the GPU memory usage of components that are parallel executed needs to be lower than the available hardware memory; moreover, the components should be independent of each other w.r.t. data communication.

In this work, we provide a method that automatically computes schemes of components that can be executed in parallel on GPU. The method is formally described and includes hardware and software specifications (Section V). Furthermore, the method provides optimized solutions considering component extra-functional properties (i.e., execution time). Resembling with the bin-packing problem and the multiprocessor scheduling problem which are combinatorial NP-hard problems [3], heuristics need to be utilized in finding solutions. For the implementation part, we use a mixed-integer non-linear programming (MINLP) heuristic approach to compute execution schemes (Section VII). As validation, we utilize an underwater robot example to describe the feasibility of our method (Section VIII).

DOI reference number: 10.18293/SEKE2017-137

II. CPU-GPU Embedded Systems

Initially, GPUs appeared in late 90s and were used only in graphics-based applications. Nowadays, the GPU is used in various types of applications such as cryptography [4] and simulation of bio-molecular systems [5], becoming a general-purpose unit.

The GPU may be seen as a complementary unit to the CPU. While the CPU is designed to rapidly execute in a sequential manner each instruction, the GPU, being equipped with thousand of computation threads, excels in parallel data processing. Therefore, CPU-GPU applications perform best when distributing the right job (i.e., sequential and parallel) to the right processing unit (i.e., CPU and GPU). For example, the massive amount of data of a vision system is processed onto GPU while the CPU takes care of e.g., histogram computations [6]. An important characteristic of the GPU is that it cannot function without the CPU. Considered the brain of a system, the CPU is the one that triggers all the activities that are executed on GPU. We need also to mention that, once an activity is started to be executed by GPU, it can not be paused or preempted.

GPU integration to embedded boards is today possible through various type of systems such as NVIDIA Jetson TK1, AMD Kabini and ARM MALI. The CPU-GPU embedded systems may increase the system performance of existing applications. For example, the stereo matching application for embedded systems has a considerable increase of frame rate processing when is performed onto GPU [7].

Another trend in embedded systems is the usage of component-based development. Through the existing component models, CBD is successfully used in industry; we mention AUTOSAR that became a standard in automotive development, Koala used by Philips and IEC 61131 used for programmable logic controllers. Another industrial component model is Rubus used by e.g., Volvo Construction Equipment branch [8]. Rubus follows the pipe-and-filter interaction style that provides a precise control flow of the system which makes Rubus applicable to particular embedded system domains (e.g., real-time systems and safety systems) [9].

Following the CBD approach, a CPU-GPU system is constructed from: *i)* regular components with CPU characteristics, implemented to be executed onto CPU; and *ii)* components with GPU functionality[1] (i.e., GPU-aware components) that have both CPU and GPU characteristics. A GPU-aware component has its functionality (or part of it) specific developed to be executed on GPU. Besides the functional description, a GPU-aware component is characterized by qualities and constraints known as extra-functional properties (EFPs). For example, a GPU-aware component may have as quality the execution time performance while as constraints, it may demand a specific number of GPU threads and memory usage.

The development of GPU-based applications is realized through different programming models and the two most known are CUDA and OpenCL. While CUDA is developed by the NVIDIA vendor to be used only for their GPUs, OpenCL is a general model developed by KHRONOS group that targets GPUs produced by e.g., INTEL, AMD and NVIDIA.

[1]During the rest of the paper, we refer to a component with GPU functionality as a GPU-aware component

III. Problem Description

For traditional CPU-based embedded systems, a way to achieve parallelism is to assign components to different e.g., CPU cores, and let the components to be executed in parallel (by the OS). We assume that there are enough resources and components are data independent of each other (i.e., do not have communication connections). Basically, the parallelism is influenced by the number of distinct processing units that a system is composed of. For example, an embedded system with a quad-core CPU can have at a given time instance a maximum of four components that can be executed in parallel.

Due to a different architecture, the way to achieve parallelism on GPU is different than on CPU. A GPU is composed of hundreds of computation cores and thousands of threads and when an activity is executed, it may consume tens of cores and thousands of threads. Therefore, the metric to reason about GPU parallelism is not the numbers of cores but is a relation between several factors such as the number of independent software activities, hardware limitation (e.g., available GPU memory) and resources demands (e.g., computation threads usage). Each GPU platform has a physical limitation regarding the number of activities that can be simultaneously executed. For example, an NVIDIA GPU with a Pascal architecture and compute capability 6.1 can run simultaneous up to 32 activities, assuming there are enough resources to sustain their execution [10].

In this work, we assume that a GPU can execute simultaneous as many components as needed due to two main reasons: *i)* it is difficult to develop an embedded system that contains e.g., 32 independent GPU-aware components; and *ii)* even if the system has a high number of independent GPU-aware components, the GPU parallelism is limited by the component usage of hardware resources. For example, in a complex system that contains 10 independent GPU-aware components that have different image processing algorithms, only few of them can be executed in parallel due to the component high resource requirements, i.e., memory, threads and registers usage. The rest of the components wait for the resources to be released in order to be executed.

IV. System Description

The section describes our vision and assumptions on the software and hardware models of a CPU-GPU embedded systems.

A. The software system

The software application is composed of several components that communicate using various styles. For example, the Rubus component model follows the pipe-and-filter architectural style, where components are seen as filters that process data while the communication between components are pipes

that transfer data. Fig. 1 presents a general component-based system with GPU functionality, that follows the pipe-and-filter style and each of its GPU-aware component is characterized by various specifications. For example, component $C1$ is characterized by its quality (i.e., execution time) and EFPs as a set of GPU-specific requirements (e.g., usage of GPU memory). Because all GPU activities are triggered by the CPU, a GPU-aware component is characterized by both CPU and GPU specific properties. As our work targets the optimization of GPU activities execution, we focus only on the GPU-aware components and their GPU-specific properties, and discard the CPU-specific properties.

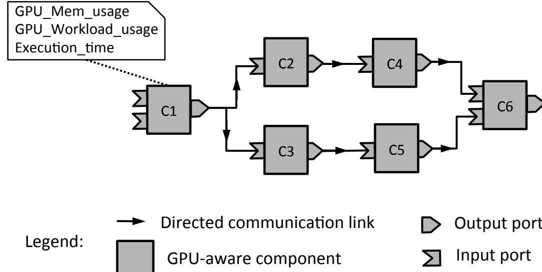

Fig. 1: System with GPU-aware components

In this initial work, we simplify and abstract the component GPU-specific properties as follows:

- the *GPU memory usage* specifies the component requirement of the GPU memory usage. The GPU has a hierarchical memory level, but we abstracted away several memory layers and use only the main memory level. The other memory layers are important factors and may be used in future work to extend our solution to a more detailed and precise software model.
- the *GPU workload* describes the GPU computation workload usage of the component; the metric utilized to describe this property is the number of computation threads. Several other factors related to the GPU workload are abstracted away (e.g., number of registers used by a thread).
- the *execution time* presents the time required by a component to fulfill its execution onto GPU.

B. The hardware system

Our focus being on GPU, we have abstracted away the CPU-specific properties (e.g., available RAM and CPU load) and characterized the hardware platform with only GPU-specific properties, as follows:

- the *available GPU memory* presents how much of GPU main memory a hardware is characterized by;
- the *GPU computation load* depicts the total of GPU computation threads a hardware is equipped with.

Similar with the software description, other influential GPU properties such as different memory levels (e.g., share memory) and the total amount of registers are removed in this introductory stage of our work.

V. Method Overview

In order to determine the components that may be executed in parallel and their execution order, the software and hardware properties are fed as input to our method. The hardware properties describe the platform resource limitation while the software specifications describe the component resource requirements and the communication pattern. The pattern of component communication has an important role when evaluating which component may run in parallel. For example, in Fig. 1, based on the communication pattern, we observe that $C2$ and $C4$ cannot be executed in parallel. $C4$ is depended of the $C2$ output and hence, they will be always sequentially executed.

The computed solution is expressed as a list composed of sublists of GPU-aware components. Each sublist contains components that can be, at a time instance, executed in parallel; the order of the sublists presents the order of the components execution.

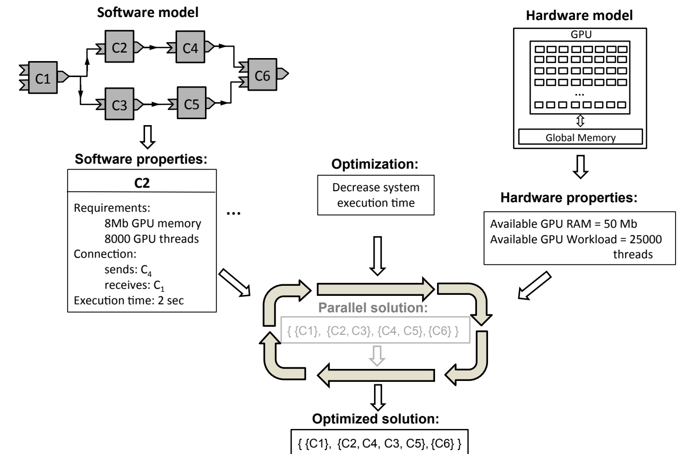

Fig. 2: The high-level overview of the proposed method

The solution overview is described in Fig. 2. Each component describes its resource requirements (e.g., GPU memory usage), how is connected to other components (i.e., to which component sends and from which component receives data) and its performance (i.e., execution time). The method calculates the components that can be executed in parallel, and their execution order.

Moreover, the method calculates an optimized parallel execution solution based on the component execution times. In order to describe the general idea of the optimization, we use the example with six connected components presented in Fig. 2. An initial solution is presented in Fig. 3(a) where, after $C1$ is executed, $C2$ and $C3$ are executed in parallel, followed by $C4$ and $C5$, and finally $C6$. The total execution time of the system is 5.8 seconds. Because $C5$ component is only dependent of $C3$ output data, and $C3$ has a shorter execution time than $C2$, an improved solution is to include $C5$ in the second batch [2] as it is seen in Fig. 3(b). This solution can be

[2]we will call the group of components that may be executed in parallel as a *batch* of components

further optimized when introducing $C4$ in the second batch, resulting an overall execution time of 5 seconds (Fig. 3(c)).

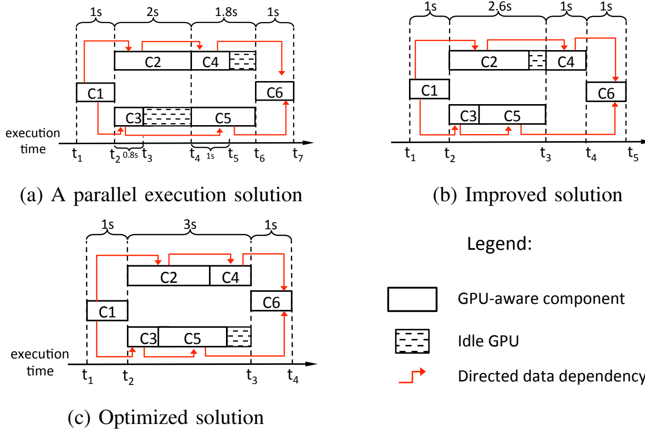

Fig. 3: Optimization of a parallel execution solution

In general, we try to reduce the system overall execution time by possibly increasing and/or decreasing the execution time of different batches and possible reducing the number of batches. More precise, we look into batches where the GPU is idle and we try to fit in suitable components from the adjacent batch. For example, in Fig. 3(b) we migrated $C5$ in batch 2 because the GPU was idle (i.e., $C3$ has a short execution time) and $C5$ is only dependent of $C3$. We notice that, in the updated second batch (Fig. 3(c)), GPU is idle after $C5$ finishes, but we cannot fit in $C6$ because it is also dependent of $C4$ and there will be no improvement gain.

The optimization idea resembles with a bin packing problem, where we have already a number of bins and the solution is improved by re-fitting the bin sizes. This type of problem, where bin sizes are changed based on their items and their connection, is NP-hard [3], i.e., an exact solution cannot be calculated in feasible time (unless $P = NP$).

VI. Method Realization

This section presents the mathematical formalization of our method starting with the description of the (software and hardware) system followed by the system constraints and initial solution calculation, and finally the optimization step.

A. System definition

1) Let be $C = \{c_1, ..., c_n\}$ a set that contains a finite number of components with GPU functionality. Each component $c_i \in C$ is characterized by two multi-valued functions $Send : C \rightarrow C$, $Rec : C \rightarrow C$, and three single-valued functions $GPU_Mem_usage : C \rightarrow \mathbb{N}_{>0}$, $GPU_Load_usage : C \rightarrow \mathbb{N}_{>0}$ and $exec_time : C \rightarrow \mathbb{Q}_{>0}$ described as follows:
   - *Send(c)* = a sublist that may contain none, one or several components that receive data from c;
   - *Rec(c)* = a sublist that may contain none, one or several components that send data to c;
   - *GPU_Mem_usage(c)* = the GPU memory usage of c;
   - *GPU_Load_usage(c)* = the number of GPU threads required by c; and
   - *exec_time(c)* = execution time of c.
2) The hardware is characterized by two constants: $GPU_Mem \in \mathbb{N}_{>0}$ and $GPU_Load \in \mathbb{N}_{>0}$, where:
   - *GPU_Mem* = the available GPU global memory;
   - *GPU_Load* = available number of GPU threads.

B. Constraints

- sum of the GPU memory usage of components from a batch (same sublist - see Section VI-C), cannot exceed the available GPU memory:

$$\sum_{c \in \{c | c \in C_i \subset C\}} GPU_Mem_Usage(c) \leq GPU_Mem$$

- sum of the GPU thread usage of components from a batch, cannot exceed the available GPU available threads:

$$\sum_{c \in \{c | c \in C_i \subset C\}} GPU_Load_Usage(c) \leq GPU_Load$$

C. Initial Solution Calculation

We see the solution as a list composed of sublists of components, i.e., $C = \{C_1, ..., C_k\}$. Each sublist contains components that can be parallelized; the sublist order presents the order of components execution. Each sublist is constructed based on its previous adjacent sublist as follows. We start by determining the first sublist that contains components with no input data; the following sublist contains components that receive data only from the components contained by the first sublist, and so on. In general, the solution list has two types of sublist elements:

- the first sublist type element $C_1 = \{c_p, ...\}$ that contains at least one element c_p and $Rec(c_p) = \varnothing$;
- the general sublist type element $C_k = \{c_q, ...\}$, where $\forall c_q \in C_k$, $Rec(c_q) \neq \varnothing$ and $Rec(c_q) \subset C_{k-1}$.

We mention that there exist only one first sublist type element while the general sublist type element may expand into none, one or several items. Some special cases may result in a solution with only one sublist item. For example, a system that contains one or several components that do not communicate among each other, will be executed in parallel (enclosed into one sublist element) if there are enough hardware resources.

D. Optimization

The initial solution calculated in the previous step is optimized (if possible) by decreasing the overall system execution. The overall idea, as described in Section V, is to look into batches where the GPU is idle and try to fit in components from an adjacent batch. Therefore, having a total of *k* batches $C = \{C_1, ..., C_k\}$, the sum of execution times of all batches is minimized:

$minimize\ ExTime = \sum_{i=1}^{k} cost_i$, where
$cost_i$ represents the cost (i.e., the execution time) of a single batch C_i and is calculated by taking the largest cost (i.e., largest component execution time) from that batch:

$cost_i = \max_{\forall c_p \in C_i}(exec_time(c_p))$.

Moreover, we may increase the cost of a batch by taking a component with (strictly) smaller cost than the batch's cost (i.e., a part of the GPU is idle) and adding to it the cost of a connected component from the adjacent batch:

$\forall c_j \in C_i,\ exec_time(c_j) < cost_i,$
$exec_time(c_j) = exec_time(c_j) + exec_time(c_q)$, where

c_q is a connected component $c_q \in C_{i+1}$,
$c_q \in Send(c_j)$ and all its connected components from batch i have a lower execution time than c_j:

$\forall c_m \in Rec(c_q) \wedge c_m \in C_i,$
$exec_time(c_m) < exec_time(c_j).$

The last condition is to ensure that the c_q component, when is added to the *i*-th batch, will not need to wait for the output data of another c_m component (from the same *i*-th batch) and will directly execute after c_j component finishes.

VII. Implementation

The optimization challenge is an NP-hard combinatorial problem [3]. Therefore, heuristic techniques need to be employed in order to find solutions. We selected the MINLP technique to address our method and to calculate feasible solutions. One solver that handles MINLP problems is SCIP [11], being one of the fastest non-commercial solvers existing on the market [12]. The solver divides the problem into smaller subproblems (know as *branching*) that are solved recursively. Moreover, for the solver to interpret our allocation model, we used the ZIMPL language [13] that translates the mathematical formulation into a readable format by SCIP. The following paragraphs briefly describe parts of our method implementation using the ZIMPL language.

Listing 1: Translation of the parallel execution model

```
1  set C := {"c1","c2","c3"};
2  param GPU_mem_use[C]:=<"c1"> 10,<"c2"> 80,<"c3"> 50;
3
4  set C_0 := { <c> in C where Rec(c) = 0 };
5  set C_1 := { <c> in C where Rec(c) in C_0 };
6
7  subto constraint: forall SubList in Sol  do
8   (sum <c> in SubList :GPU_mem_use[c]) <= GPU_mem_available;
9
10 minimize gpu_cost: sum SubList in Sol :
11  forall <c> in SubList do max(exec_time(c));
```

Besides the actual translation of the model constraints, it is required to construct a system model (in ZIMPL) in order to execute the model and find solutions. The system construction can be achieved in two ways: *i)* hard-coding the system and its characteristics directly into the ZIMPL program; and *ii)* reading the specifications from a file. The former mean is illustrated in Listing 1, where a system is defined as a set *C* of three components (line 1); the next line captures the GPU memory usage of each component. Similarly, we define the rest of the system specifications such as the available GPU memory and the component execution times.

We continue by constructing the initial solution that comprises of sublists of components. The first sublist *C_0* contains components that have no input data (see Section VI-C - first sublist type element); the following sublist, *C_1* is constructed based on the *C_0* content, i.e., all the components that receive data only from components of *C_0* sublist (lines 4 and 5). Regarding the implementation of constraints, we present the memory constraint, where the sum of the GPU memory usage of components from a sublist is less (or equal) than the available hardware memory (lines 7 and 8). The last part of the Listing describes a reduced form of the optimization function (due to the complexity structure), where we calculate the cost of each sublist *SubList* from the initial solution *Sol*, and minimize their summed cost (lines 10 and 11).

VIII. Running Case Evaluation

We examine our proposed method through a feasibility evaluation of an underwater robot demonstrator. The robot contains a CPU-GPU embedded board that communicates with actuators (e.g., thrusters) and sensors (e.g., cameras) [14]. Using the continuous feedback provided by cameras, the robot autonomously navigates underwater in fulfilling various missions (e.g., tracking red buoys). For designing the robot architecture, we used the Rubus component model [15] due to its fitting for developing streaming-of-event type of applications.

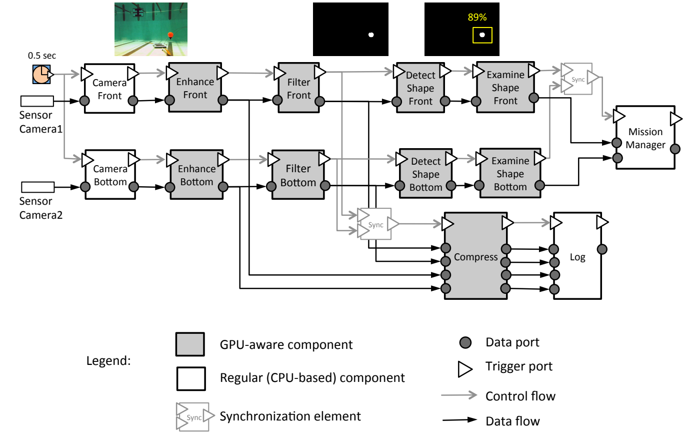

Fig. 4: The Rubus architecture of the vision system

Fig. 4 presents the vision system architecture of the robot. The physical (front and bottom) cameras provide raw data that is converted into readable frames by *CameraFront* and *CameraBottom*. These frames are forwarded to *EnhanceFront* and *EnhanceBottom* that improve the frames quality, by e.g., removing harsh edges. *FilterFront* and *FilterBottom* filter images using red-color criterion, resulting black-and-white frames. *DetectShapeFront* and *DetectShapeBottom* detect the shapes of the objects from the frames (e.g., circle, square) and forward their results to *ExamineShape* components that verify the found shapes against predefined shapes. The findings are sent to *MissionManager* that takes appropriate decisions. The *Compress* component compresses frames (e.g., resize) to allow

the (CPU) *Log* component to keep trace of the robot navigation for debugging purposes.

Each frame produced by cameras has a number of 600*400 pixels. When a frame is processed, we set that a GPU thread operates on 32 pixels. The GPU hardware has a total of 65536 threads and 128 Mb of memory. After translating the vision system using ZIMPL language and applying our proposed model on it, we obtain a solution illustrated by Table I.

TABLE I: The vision system execution scheme

Sublist (Batch)	GPU-aware component	Memory usage(Mb)	Thread usage	Execution time(ms)
1	Enhance Front	1.2	8000	35
	Enhance Bottom	1.2	8000	35
2	Filter Front	1.2	8000	30
	Filter Bottom	1.2	8000	30
3	DetectShapeFront	1.2	8000	45
	ExamineShapeFront	1.2	8000	45
	DetectShapeBottom	1.2	8000	40
	ExamineShapeBottom	1.2	8000	40
	Compress	4.8	24000	70

The solution orders the components into three batches of execution. We notice that the solution is optimized as batch 3 contains five components to be executed in parallel. An intermediate solution contains four batches, where in batch 3 only *DetectShapeFront*, *DetectShapeBottom* and *Compress* would be executed in parallel. Due to the high execution time of *Compress* and the hardware having enough resources, *ExamineShapeFront* and *ExamineShapeBottom* are migrated into batch 3.

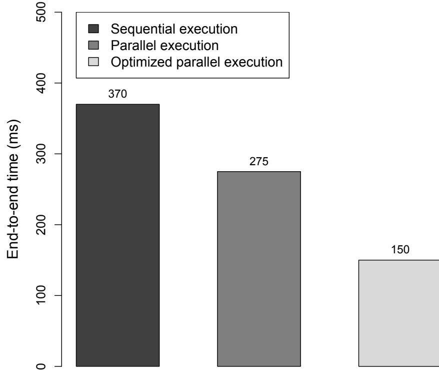

Fig. 5: Different execution times of vision system

Regarding the performance of the vision system, Fig. 5 presents the execution times when the system is executed in three cases. When the system is executed sequentially, the performance is the worst (i.e., 370 ms). An intermediate solution that executes the system in four batches, improves the system performance (i.e., 275 ms). Optimizing the solution and executing the system in three batches offers an enhanced performance (i.e., 150 ms).

By applying out method on the vision system case study, we improved the overall system performance. The specifications of our system allowed the method to provide an optimized solution.

IX. Related Work

Embedded systems embraced heterogeneity to improve system performance. Nowadays, we have platforms with multicores (e.g., SoC quad-core ARM processor [16]) and GPUs (e.g., AMD Carrizo APU). To manage the specifics of the new platforms, CBD introduced ways to handle the hardware particularities. We mention the work of Kopetz et al. [17] that explores the design alternatives of the AUTOSAR component model when targeting multi-core ECUs. The CPU-FPGA platforms are addressed by Andrews et al. [18] that introduce a way to utilize COTS components, by synchronizing CPU-FPGA computations inside the components. The Rubus component model is extended with new artifacts (e.g., GPU ports), to allow efficiently development of CPU-GPU embedded systems [19].

Due to an increased complexity and challenging quality requirements, system optimization approaches has proliferated. There is a body-of-knowledge presented in different surveys [20] [21], that targets optimization of software architectures from different system domains (e.g., embedded systems and information systems) considering various quality attributes. Yet, there is a reduce amount of work that addresses the optimization of component-based architectures of embedded systems with different computation nodes. We mention the work of Campeanu et al. that targets component-based systems with many CPU and GPU computation nodes [22]. The authors allocate components over hardware considering their (CPU and GPU) requirements and optimize the allocation based on different criteria. The drawback of this work is that it does not take in attention the architecture design and considers that all GPU-aware components can be parallelized at once. In our work, we consider how the components are connected and optimize their parallel execution on a single GPU node.

Specific scheduler solutions for GPU-based systems are addressed by various works. We mention Muyan-Özçelik et al. [23] that developed several scheduling algorithms for (GPU) tasks, where a task is defined as a series of operations, i.e., a host-to-device copy, a GPU (kernel) execution and a device-to-host copy. For example, an algorithm increases the system performance by scheduling copy operations in the same time as GPU (kernel) executions. The work presents the findings of its schedulers using NVIDIA technology. By targeting only NVIDIA GPUs, some of their claims are based on specific hardware technology which may not hold for GPUs from other vendors. Moreover, they have a high control over scheduled tasks. In our work, one or several tasks may be incorporated into a component and we do not have the same level of control over them. For example, we do not know when a device-to-host copy operation (encapsulated into a component) ended in order to give the control to a GPU (kernel) execution (encapsulated by another component).

Moreover, even if we create such mechanisms, the context-switching between components would be very expensive. Basically, it would worth to have these mechanisms when components would have very large copy operations and GPU (kernel) executions. Other works use the same (refined) control level in order to provide parallelization strategies. We mention a two-level parallelization strategy that works directly with the GPU (kernel) functionality by e.g., analyzing its loop iterations and their statements [24].

As our optimization challenge is similar to the bin-packing problem, we want to mention the surgical scheduling problem, where the operating rooms seen as bins, can change their (time) capacity by increasing the number of operations [25]. In addition, we find the multiprocessor scheduling problem [26] related to our challenge and we refer specifically to the Gang scheduling [27] which is considered to be an efficient algorithm for parallel and distributed systems. One of its types is Bags of Gangs (or Bags of Tasks) [28] in which the jobs, considered as independent gangs that belong to a bag, are sent to be executed by the system. A bag finishes its execution only when all of its gangs finish.

X. Conclusions

This work introduces an initial method that addresses the parallel execution of components on GPU. Our proposed method computes execution schemes by considering: *i)* hardware characteristics (e.g., available GPU memory); *ii)* software constraints (e.g., required number of GPU threads); and *iii)* component communication pattern. The method optimizes the computed schemes w.r.t. performance (i.e., execution time) resulting in schemes with maximum degree of component parallelism. Being an NP-hard combinatorial problem, the optimized schemes are calculated by using a MINLP heuristic method. The last part of our work presents the feasibility aspect of the proposed method when is applied on an underwater robot case study.

To the best of our knowledge, there are no developed component mechanisms to execute GPU-aware components in parallel. In addition, as several components may simultaneously access the GPU, mechanisms to protect the GPU (seen as a shared resource in this context) need also to be developed. We consider these aspects as future directions of our work.

Acknowledgments

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

References

[1] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*, 2002.

[2] T. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge," in *Proceedings of the 14th International Symposium on Formal Methods*, 2006.

[3] S. Anily, J. Bramel, and D. Simchi-Levi, "Worst-case analysis of heuristics for the bin packing problem with general cost structures," *Operations research*, 1994.

[4] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *IEEE International Conference on Signal Processing and Communications. ICSPC 2007.*

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

[6] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "GPU-based video feature tracking and matching," in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[7] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze, "A fast stereo matching algorithm suitable for embedded real-time systems," *Computer Vision and Image Understanding*, 2010.

[8] Arcticus Systems, "Customers," http://www.arcticus-systems.com/links/, accessed: 2017-03-01.

[9] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, 2011.

[10] "NVIDIA CUDA C programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: 2017-03-01.

[11] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig, "The SCIP optimization suite 3.2," ZIB, Tech. Rep., 2016.

[12] H. Mittelmann, "Mixed integer linear programming benchmark," http://plato.asu.edu/ftp/milpc.html, accessed: 2017-03-01.

[13] T. Koch, "Rapid mathematical prototyping," Ph.D. dissertation, Technische Universitat Berlin, 2004.

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

[15] Arcticus Systems, "Rubus models, methods and tools," http://www.arcticus-systems.com, accessed: 2017-03-01.

[16] ARM, "The ARM Cortex-A53 processor," https://www.arm.com/products/processors/cortex-a/, accessed: 2017-03-01.

[17] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber, "Automotive software development for a multi-core system-on-a-chip," in *Workshop of Software Engineering for Automotive Systems*, 2007.

[18] D. Andrews, D. Niehaus, and P. Ashenden, "Programming models for hybrid CPU/FPGA chips," *Computer*, 2004.

[19] G. Campeanu, J. Carlson, S. Sentilles, and S. Mubeen, "Extending the Rubus component model with GPU-aware components," in *19th Int. Symposium on Component Based Software Engineering*, 2016.

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

[21] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, 2004.

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

[23] P. Muyan-Özçelik and J. D. Owens, "Multitasking real-time embedded GPU computing tasks," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2016, pp. 78–87.

[24] J. Shirako, A. Hayashi, and V. Sarkar, "Optimized two-level parallelization for GPU accelerators using the polyhedral model," in *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 2017, pp. 22–33.

[25] J. H. May, W. E. Spangler, D. P. Strum, and L. G. Vargas, "The surgical scheduling problem: Current research and future opportunities," *Production and Operations Management*, 2011.

[26] M. R. Gary and D. S. Johnson, "Computers and intractability: A guide to the theory of NP-completeness," 1979.

[27] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *ICDCS*, 1982.

[28] W. Cirne, F. Brasileiro, J. Sauve, N. Andrade, D. Paranhos, E. Santos-neto, and R. Medeiros, "Grid computing for bag of tasks applications," in *In Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment*, 2003.