Developing CPU-GPU Embedded Systems using Platform-Agnostic Components

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

Abstract—**Nowadays, there are many embedded systems with different architectures that have incorporated GPUs. However, it is difficult to develop CPU-GPU embedded systems using component-based development (CBD), since existing CBD approaches have no support for GPU development. In this context, when targeting a particular CPU-GPU platform, the component developer is forced to construct hardware-specific components, which are problematic to (re-)use in different contexts. Moreover, hard-coding specific GPU-usage characteristics (e.g., the number of utilized GPU threads) inside the component is not possible without making detailed assumptions about the system in which the component is used, which conflicts with separation-of-concerns CBD principle.**

The paper presents a solution to allow component-based development of platform-agnostic CPU-GPU embedded systems through: *i)* high-level API, *ii)* adapters, and *iii)* code template. The API abstracts the specifics of the different platforms, while the adapters externalize hardware-specific activities outside components. We also raise the decision regarding the GPU-usage specifications, from the component to the system level. Furthermore, to minimize the development effort, we provide a code template that contains ready-made code fragments required for GPU development. As a case study, we examine the feasibility of our solution applied on a component-based vision system of an underwater robot.

I. Introduction

One of the challenges of many modern embedded systems is to process, with sufficient performance, the huge amount of data originated from the interaction with the environment. One solution to this problem comes from the usage of general-purpose Graphics Processing Units (GPUs), that have a parallel execution model and can process data in a parallel manner. Today, various embedded platforms with GPUs are available with very different architecture characteristics in terms of physical resources (e.g., computation threads), size, support and energy consumption. For example, systems with very high processing demands might use a Condor GR2 GPU that provides huge computation power, but comes with a high energy consumption (i.e., up to 69.3 Watt), while battery-powered devices might have to settle for the reduced computation power of a Mali-470 GPU due to its much lower energy consumption (i.e., 1 Watt).

One way to develop embedded systems is the usage of component-based development [1]. CBD advocates the efficient development of applications by composing already existing software units called components. A benefit of the CBD is the ability to (re-)use the same component for different applications and contexts, thus boosting the development efficiency. The CBD methodology is successfully adopted by industry through various component models such as AUTOSAR, IEC 61131 and Rubus.

When it comes to development of CPU-GPU embedded systems, the existing CBD approaches have no GPU support. Using these approaches, the component developer needs to encapsulate hardware-specific activities inside the components to address a single CPU-GPU platform. For example, a component needs to encapsulate different memory access mechanisms when executed on GPU Condor GR2 and Mali-G71 due to their different physical construction. This way of constructing platform-specific components reduces the component ability to be (re-)used in other contexts. Moreover, the component developer needs to encapsulate inside the components, GPU-usage settings (e.g., number of utilized GPU threads) which is problematic without information about the rest of the system (e.g., how many other components utilize the GPU). Encapsulating aspects that require system-level knowledge, conflicts with the separation-of-concerns CBD principle.

In this paper, we introduce a solution that allows the component developer to construct lightweight platform-agnostic CPU-GPU systems, by using: *i)* a *high-level API*, *ii) adapters*, and *iii)* a *code template*. The API abstracts the different characteristics of CPU-GPU platforms allowing development of platform-agnostic components. The decision regarding the GPU-usage settings (e.g., GPU threads utilization) is raised from the component to system level which improves important component properties, such as reusability and maintainability. Furthermore, we allow the development of lightweight components by automatically generating efficient memory management activities between communicating components, encapsulated inside adapters. Additionally, we minimize the development effort through a code template that provides ready-made code-snippets necessary for GPU development.

II. Background

Today, the technological improvements allow the GPU integration onto embedded platforms. There are two types of platforms: *i)* discrete GPU (*dGPU*), and *ii)* integrated GPU (*iGPU*). A dGPU has its private memory systems and data, in order to be processed, needs to be copied on it (with additional overhead) via e.g., a PCIexpress bus. An iGPUs shares the same physical chip with a CPU. There are several

variation of iGPUs. Some systems with iGPU may have their memories divided into two distinct parts, one for GPU and another for CPU. In this case, there is still need for data transfer activities but the copy overhead is minimized due to the physical location (i.e., on the same memory chip). A more advanced iGPU architecture offers concurrent access to the same memory to both the CPU and GPU, without the need of data copies.

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

Nowadays, various programming models allow the development of GPU applications. The two most prominent models are CUDA and OpenCL. They are designed to work with programming languages such as C and C++. Being developed by NVIDIA, CUDA only targets NVIDIA GPUs. On the other hand, OpenCL is a general framework that supports different types of computational units such as CPUs, GPUs and field-programmable gate arrays (FPGAs). In this work, we focus on OpenCL as it is supported by multiple platforms and vendors (e.g., ARM, AMD, Altera, IBM, INTEL, Samsung, Xilinx).

Depending on their construction, different embedded platforms support distinctive OpenCL versions. Until today, there are six existing OpenCL versions (i.e., 1.0, 1.1, 1.2, 2.0, 2.1 and 2.2). Different technological improvements are handled by different OpenCL versions. For example, while the 1.1 version supports only distinct address spaces, the 2.0 version supports different memory architectures. However, even if OpenCL 2.0 supports shared memory, not all platforms (with OpenCL 2.0 support) are physical equipped with shared memory. For example, ARM Mali-G71 architecture supports only shared virtual memory and distinct address spaces, but does not support shared memory due to its physical construction.

III. The Overall Challenge

Nowadays, no existing component model provides explicit support for GPUs. Because component models are at the core of CBD, this issue hinders appropriately using CBD for the development of CPU-GPU embedded systems. As a consequence, when using the currently existing CBD approaches, the component developer is required to develop and encapsulate inside components, besides the functionality, a GPU environment through which the GPU can be accessed. In order to set up the GPU environment, the developer needs to have information about: *i)* the target platform, e.g., hardware capabilities; and *ii)* the system in which the component will be used, e.g., software architecture.

To address the different characteristics of various CPU-GPU architectures, the developer needs to build hardware-specific components. For example, a component that would be executed on a dGPU must have means to perform transfer data (through specific operations) between the distinct (CPU and GPU) address spaces of the platform, while another component that targets platforms with iGPUs (with a full address space shared with the CPU), does not require any data transfer activities. Moreover, the developer encapsulates particular settings regarding the usage of the GPU platform, by making assumptions about the system in which the component will be used. For example, it is challenging to set up inside the component the number of computational threads to be used by the GPU without system informations such as the total amount of available GPU threads, the overall system architecture and which other components use the GPU.

The existing GPU-related component development practice is a complex process, error-prone and time consuming. It decreases the component maintainability and makes it difficult to (re-)use a component on different CPU-GPU hardware architectures. Moreover, the existing practice brakes the CBD separation of concerns principle between component and system development.

To make the use of GPU compatible with a CBD approach, the main following challenges must be addressed: *i)* the platform-dependency; *ii)* the GPU configuration done at the component level; and *iii)* unrelated code w.r.t. the component functionality (e.g., GPU environment code).

IV. Development of Platform-Agnostic CPU-GPU Embedded Systems

In this work, we consider white-box components (e.g., Rubus and IEC 61131 components), that are units of source code, readable and directly modifiable by the component developer. Moreover, due to the embedded systems targeted in this work (e.g., real-time, control-type systems), we place our work in the context of *pipe-and-filter* architecture style due to its feasibility for this embedded environment. In this context, through our proposed solution, we: *i)* allow the developer to build generic components with GPU functionality; *ii)* raise the component GPU configuration activity to the system level; and *iii)* minimize the code manually introduced by the component developer.

We introduce a *high-level API* to conceal the differences in the characteristics of the existing CPU-GPU architectures. The API contains a set of functions that allows development of generic components with GPU functionality, that can target any of the existing CPU-GPU platforms. For example, one aspect that the high-level API handles is to abstract the different data transfer operations used by different platforms.

We also externalize outside components, *specific settings* required by the GPU functionality. In this way, the decisions regarding component GPU resource utilization, are taken at the system level by the component integrator which may improve the overall system efficiency, and different properties such as reusability and maintainability. Furthermore, to minimize the hand-written GPU code of the developer, we provide a component *code template*. The template consists of ready-made code fragments that are necessary when developing GPU functionality.

In order to allow components with GPU functionality to be used with no changes on different platforms, we build upon

our previous work [2] [3] and introduce automatically generated *adapters*. Instead of encapsulating inside components, different (data transfer) operations that are characteristic to different CPU-GPU platforms, we automatically externalize and encapsulate them inside transparent adapters.

More details about the platform-agnostic component solution are presented in the following subsections.

A. High-Level API

To facilitate the platform-agnostic development of components with GPU functionality, we provide a high-level API that abstracts the different characteristics of existent hardware platforms. The API is composed of functions that transparently call the OpenCL mechanisms that correspond to the utilized platform. We introduce four functions provided by the API, as they are the minimum required for building an OpenCL application that can target various platforms:

- *apiCreateBuffer* to allocate GPU memory;
- *apiReleaseBuffer* to deallocate GPU memory;
- *apiTransferBuffer* to transfer data; and
- *apiSetKernelArg* to set up the parameters for GPU functions.

To illustrate the construction of the API, we present in Listing 1, the *apiCreateBuffer* function. The 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 4). 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*).

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.

Listing 1: The apiCreateBuffer function

```
1   void *apiCreateBuffer(cl_context context, cl_mem_flags flags,
          size_t size, void *host_ptr, cl_int *errcode_ret)
2   {
3   #if !defined(CL_VERSION_2_1) && !defined(CL_VERSION_2_0) && (
          defined(CL_VERSION_1_2) || defined(CL_VERSION_1_1) )
4    return (void *)clCreateBuffer(context, flags, size, host_ptr,
           errcode_ret);
5   #endif
6
7   #if defined(CL_VERSION_2_0) && defined(CL_VERSION_2_1)
8    cl_device_svm_capabilities caps;
9    cl_int svm=clGetDeviceInfo(deviceID,CL_DEVICE_SVM_CAPABILITIES,
           sizeof(cl_device_svm_capabilities),&caps,0);
10
11   if (svm==CL_SUCCESS && (caps & CL_DEVICE_SVM_FINE_GRAIN_SYSTEM))
12    return malloc(size);
13   else if (svm==CL_SUCCESS && (caps & CL_DEVICE_SVM_COARSE_GRAIN))
14    return clSVMAlloc(context, flags, size, unsigned int alignment);
15   else if (svm == CL_INVALID_VALUE )
16    return clCreateBuffer(context,flags,size,host_ptr,errcode_ret);
17  #endif
18  }
```

B. Code Template and GPU Settings

When developing a component with GPU functionality, the component developer needs to build the GPU environment of the component. In more details, the component developer creates a GPU *context* to manage the platform resources, and a software mechanism (known as the *command queue*) to send commands to the GPU. Once the component's GPU functionality (known as the *kernel*) is constructed, the component developer decides how many GPU threads to use when executing the functionality. After dispatching the *kernel* to be executed by the GPU, the resources are released.

To decrease the developer load, allowing to focus on the important parts of the component development, that is, the functionality, we provide a *code template* that is illustrated in Listing 2. The template provides ready-made code fragments regarding the GPU environment of the component. The uppercase bold texts pinpoint the locations where the developer needs to provide code. For example, the developer needs to introduce the component GPU functionality at line 2. The developer needs also to provide specifications for different variables, such as the size of the allocated memory (line 4) and the pointer to a variable to be released (line 24).

Listing 2: Template for components with GPU functionality

```
1   /* define the GPU functionality*/
2   const char *source = "<GPU FUNCTIONALITY>";
3   /* Create memory buffer to hold the result */
4   void *result = apiCreateBuffer(settings->contex,
          CL_MEM_WRITE_ONLY, <SIZE> ,NULL,NULL);
5   /* Create a program from the kernel source */
6   cl_program program = clCreateProgramWithSource(settings->contex,
           1, (const char **)&source,NULL,NULL);
7   /* Build the program */
8   clBuildProgram(program,1,&(settings->device_id),NULL,NULL,NULL);
9   /* Create the OpenCL kernel */
10  cl_kernel kernel = clCreateKernel(program,<NAME OF GPU
          KERNEL>,NULL);
11  /* Set the arguments of the kernel */
12  apiSetKernelArg(kernel,0, <SIZE>, <POINTER>);
13  apiSetKernelArg(kernel,1, <SIZE>, <POINTER>);
14  /* settings - GPU threads usage */
15  size_t global[2] = {(settings->global1), (settings->global2)};
16  size_t local[2] = {(settings->local1), (settings->local2)};
17  /* Execute the OpenCL kernel */
18  clEnqueueNDRangeKernel(settings->cmd_queue, kernel, 2, NULL,
          global, local, 0, NULL, NULL);
19  /* Clean up */
20  clFlush(settings->cmd_queue);
21  clFinish(settings->cmd_queue);
22  clReleaseKernel(kernel);
23  clReleaseProgram(program);
24  apiReleaseBuffer(<POINTER>);
```

Moreover, being common to all components with GPU functionality, we extracted the parts that create the *context* and *command queue* mechanisms, and generate them automatically, outside the components. We provide the means to access these mechanisms through specific settings that are automatically provided by the template. Listing 2 presents, with lowercase bold texts, the `settings->context` and `settings->cmd_queue` settings that are automatically provided by the template. In the same way, we externalized outside the component, the decision of how many GPU threads should be utilized. Settings that contain information on resource utilization are provided to each component with GPU functionality. These are depicted in the listing by the e.g., `settings->global1` and `settings->local1` settings (lines 15 and 16). The `global[2]` variable, called as the *global work size*, specifies through its two-dimensional capacity, the total number of utilized GPU threads, while

local[2], known as the *local work size*, describes the thread capacity of working (thread) groups.

C. Generated Adapters

We increase the support provided to the component developer by decreasing even more the code required to be manually written. In this sense, we capture hardware-specific memory management operations that characterize a component with GPU functionality, and externalize them outside the component. The externalized operations are encapsulated in artifacts called adapters.

When components are integrated to construct a system, the adapters are automatically generated in a transparent manner. Adapters are generated whenever different data ports (i.e., CPU and GPU data ports) of different components are connected [2] [3]. Depending on the hardware architecture, there are different cases when adapters are employed, as follows. For hardware platforms with distinct address spaces, the adapters contain specific memory operations that copy data between the address spaces. For more advanced platforms that support shared virtual memory, there is a need of a CPU-to-GPU adapter to specifically target (i.e., allocate and copy data onto) the SVM space. There is no need for a GPU-to-CPU adapter because all components can access directly data from the SVM space. For platforms with full shared memory, there is no need to utilize adapters, as all components (with or without GPU functionality) can access directly the shared memory.

V. Evaluation

This section presents the feasibility evaluation of the vision system of an underwater robot case study. The robot autonomously navigates under water with the purpose to fulfill various missions such as identifying red buoys [4]. The hardware platform contains of an electronic board with a CPU-GPU architecture that is connected to various sensors (e.g., two cameras) and actuators (e.g., thrusters). The continuous flow of data produced by the cameras, is processed by the robot's vision system using the GPU.

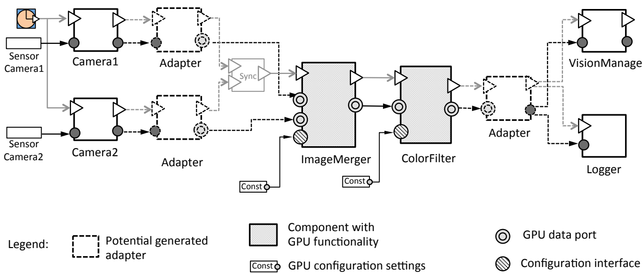

Fig. 1: Platform-agnostic vision system

Fig. 1 shows a simplified version of the component-based vision system that is designed using the Rubus component model. Two *Camera* components fetch data from physical camera sensors and forward them to *ImageMerger* that merges and reduces the noise of the two received frames using the GPU. The resulted frame is red-color filtered (using the GPU) by *ColorFilter* that provides a black-and-white image. The *VisionManager* takes appropriate actions based on the findings (e.g., shape and position of the detected object) from the frame. To record the robot's underwater journey e.g., for debugging purposes, the *Logger* component registers all the filtered frames.

The system is automatically populated with adapters that transfer data between address spaces. They are represented as dash-line artifacts as: *i)* they are automatically generated in a transparent manner, and *ii)* depending on the platform characteristics, they (or some of them) are not required (i.e., generated).

Using our solution, we implemented the *ImageMerger* and *ColorFilter* as generic components with GPU functionality. Regarding the development effort, Table I presents the size of the developed vision system. Besides the characters contained by the template (i.e., 736 characters for each component), the developer needs to manually introduce (including the GPU functionality) 1195 characters for *ImageMerger* and 859 characters for *ColorFilter*. The table describes also the number of characters automatically generated through the adapters. In Case 1, i.e., platforms with distinct address spaces, there are generated a number of three adapters (see Fig. 1) using 1110 characters. For platforms with shared virtual memory (Case 2) where there is only need of CPU-to-GPU adapters, there are generated two adapters using 584 characters. For Case 3 (i.e., platforms with shared memory) there is no need to generate adapters.

TABLE I: Manually written and generated characters

Component name	Characters				
	Template	Manually written	Automatic generated adapters		
			Case 1 DAS	Case 2 SVM	Case 3 SM
ImageMerger	736	1195	756	584	0
ColorFilter	736	859	354	0	0

DAS - Distinct Address Spaces platform
SVM - Shared Virtual Memory platform
SM - Shared Memory platform

We notice that, for platforms with distinct address space, the developer needs to construct 45% of (GPU-related part of) the vision system, while our solution provides the rest of the 55%. For platforms with shared memory (i.e., no need for adapters), the developer constructs 60% of the system, while our solution provides 40%.

VI. Related Work

In a race to satisfy the stringed demands of modern applications, embedded systems embraced heterogeneity. The PEPPHER framework proposes a component model that allows efficient hardware utilization [5]. A PEPPHER component (i.e., annotated software module) can have several variant, each with its own set of properties (e.g., performance, execution platform). A suitable variant is chosen at runtime. To increase programmability for GPUs, the framework allows component developer to use SkePU [6] skeletons (e.g., map, reduce).

Moreover, SkePU uses special artifacts (i.e., smart containers) to e.g., optimize communication and perform memory management between CPU and GPU distinct memories. In our work, we improve component aspects (e.g., reusability) by allowing the developer to use an API, a similar idea to SkePU skeletons. Much alike to smart containers, our adapters can be considered high-level memory management elements. The PEPPHER framework is limited in considering only systems where GPUs have their own private memory.

Using model-driven engineering (MDE), Rodrigues et al. provide resources to non-specialists to develop applications for GPU-based systems [7]. Basically, the developer models GPU functionality where all the GPU data resides on the GPU address space. Being developed in 2013, the work conforms with the functionality provided by the OpenCL 1.0 version. Due to the latest development evolution both in hardware and software terms, the work is limited in covering the recent (hardware and software) advancements. In the same MDE context, we want to mention the framework proposed by Gamatie et al. [8] that uses the MARTE profile to design SoC systems, and allows automatically generation of code (for e.g., formal verification, simulations). Similarly to our adapters purpose, the authors use connector elements (i.e., tilers) to adapt exchanged data size (e.g., modify a 2-dimension to a scalar array) between linked components.

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 optimization of different graph algorithms [9]. Built on top of CUDA, the API targets only NVIDIA-based GPU platforms. OpenVIDIA [10], an API that provides function calls to run common vision algorithms, abstracts OpenGL calls needed for vision processing.

There are existing technologies that support development of data-parallel applications for heterogeneous platforms, and some of them facilitate the development of applications by e.g., abstracting hardware characteristics. We mention the C++ AMP library[1] that accelerates execution of C++ code. It may, however, only be employed by a limited class of embedded systems due to its prerequisites (e.g., Windows 8 OS). AMD Bolt[2] is a C++ template library that supports only specific AMD processing units, thus limiting its usage by embedded systems with hardware from other vendors. Moreover, for particular embedded system types (e.g., real-time and safety-critical) where resources are carefully distributed, Bolt may not be an appropriate alternative due to its way of requesting system resources at runtime. Sycl is an open standard from Khronos, that builds on top of OpenCL and allows development of C++ applications for heterogeneous systems[3]. Although these technologies do not provide CBD support, they may be employed in our solution by, e.g., replacing the introduced API.

[1]https://msdn.microsoft.com/en-us/library/hh265136.aspx

[2]https://github.com/HSA-Libraries/Bolt

[3]https://www.khronos.org/sycl

VII. Conclusions

This work provides a way to develop platform-agnostic component-based CPU-GPU systems. We introduce a high-level API that abstracts the distinct hardware characteristics; using it, the developer constructs generic components that can target any CPU-GPU platforms. In addition, we lift component GPU-usage information, from the component to the system level, which improves the component maintainability and reusability. Moreover, we introduce a code template that provides ready-made code fragments required for GPU development.

As there are no developed mechanisms to allow components to be executed in parallel onto GPU, we aim to address this concern as future work.

Acknowledgments

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

References

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

[2] G. Campeanu, J. Carlson, and S. Sentilles, “A GPU-aware component model extension for heterogeneous embedded systems,” in *The Tenth International Conference on Software Engineering Advances ICSEA 2015, 15-20 Nov 2015, Barcelona, Spain*, 2015.

[3] G. Campeanu, J. Carlson, S. Sentilles, and S. Mubeen, “Extending the Rubus component model with GPU-aware components,” in *Component-Based Software Engineering (CBSE), 2016 19th International ACM SIGSOFT Symposium on*. IEEE, 2016, pp. 59–68.

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

[5] U. Dastgeer, L. Li, and C. 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:*. IEEE, 2012, pp. 711–720.

[6] U. Dastgeer, J. Enmyren, and C. W. Kessler, “Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems,” in *Proceedings of the 4th International Workshop on Multicore Software Engineering*. ACM, 2011, pp. 25–32.

[7] A. W. O. Rodrigues, F. Guyomarc’H, and J. L. Dekeyser, “An MDE approach for automatic code generation from UML/MARTE to OpenCL,” *Computing in Science & Engineering*, vol. 15, no. 1, pp. 46–55, 2013.

[8] A. Gamatie, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, “A model-driven design framework for massively parallel embedded systems,” *ACM Trans. Embed. Comput. Syst.*, pp. 39:1–39:36, 2011.

[9] Z. Fu, M. Personick, and B. 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*. ACM, 2014, pp. 1–6.

[10] J. Fung and S. Mann, “OpenVIDIA: Parallel GPU computer vision,” in *Proceedings of the 13th Annual ACM International Conference on Multimedia*. ACM, 2005, pp. 849–852.