A GPU-aware Component Model Extension for Heterogeneous Embedded Systems

Gabriel Campeanu, Jan Carlson and Séverine Sentilles

Mälardalen Real-Time Research Center
Mälardalen University, Sweden
Email: {gabriel.campeanu, jan.carlson, severine.sentilles}@mdh.se

***Abstract*—One way for modern embedded systems to tackle the demand for more complex functionality requiring more computational power is to take advantage of heterogeneous hardware. These hardware platforms are constructed from the combination of different processing units including both traditional CPUs and for example Graphical Processing Units (GPUs). However, there is a lack of efficient approaches supporting software development for such systems. In particular, modern software development approaches, such as component-based development, do not provide sufficient support for heterogeneous hardware platforms. This paper presents a component model extension, which defines specific features for components with GPU capabilities. The benefits of the proposed solution include an increased system performance by accelerating the communication between GPU-aware components and the possibility to control the distribution of GPU computation resources at system level.**

Keywords–Embedded Systems; Component-based Development; Heterogeneous CPU-GPU Systems; GPU Component Model.

I. INTRODUCTION

In the last years, various embedded computing technologies have emerged due to the rapid advance of microprocessing technology. Homogeneous single-core CPU systems have evolved into heterogeneous systems with different processing units such as multi-core CPUs or GPUs. Taking benefits of the increased computational parallel power, new applications have emerged while others improved their performance. Examples of systems that now use GPU processing hardware include vehicle vision systems [1] and autonomous vision-based robots [2]. However, a GPU is a different hardware unit that has its own memory system. As a result, combining a GPU with a CPU leads to an increase of the software complexity and the need to optimize the use of the available resources.

One way of addressing the increasing system complexity is through component-based development (CBD). In CBD, complex software applications are built by composing already existing software solutions (i.e., software components), resulting in increased productivity, better quality and a faster time-to-market. The approach has been successfully used in other domains, but has recently attracted attention also for developing software for embedded systems, as evident by industrially adopted component models such as Autosar [3] and Rubus [4]. In order to address the specifics of embedded systems, many component models targeting this domain follow a *pipe & filter* architectural style. Using this style, the components are passive and the transfer of data and control is defined statically by how they are connected, rather than the typical object oriented style with active components and method calls [5].

Having no component model support for GPU development, each component that needs to use the GPU must encapsulate various GPU specific operations such as memory initialization, and operations to shift data between the CPU and GPU memory systems. This introduces a communication overhead among components (i.e., leading to longer response times) and unnecessary code duplication. Also, these components have to encapsulate all the GPU settings required to meet their functionality. For example, each component independently decides how much GPU computation resources it uses (e.g., number of threads), which can result in a suboptimal GPU usage in the system as a whole, decreasing the system performance. The lack of efficient development methods affects several component properties (e.g., granularity, reusability), making difficult to fully use the benefits of GPU systems.

In this paper, we describe how the problem can be tackled by proposing a GPU-aware component model extension where components are equipped with GPU ports that allow component communication directly through the GPU environment. We also provide a way for the system to decide the GPU settings (e.g., number of threads) for each GPU-aware component. Enhancing the communication and delegating the component GPU settings to the system level, result in increased system performance while keeping the key benefits of the CBD approach.

The rest of the paper is organized as follows. Section II gives a background depiction of existing component model challenges in addressing efficiently the GPU hardware. A high level descriptions of the GPU-aware components is covered by Section III, where the specification of the component interfaces and GPU ports are described in depth. Section IV describes a running example which illustrates the underlying details of our solution. In Section V, a series of experiments were carried out to evaluate the performance efficiency of the proposed method. Related work is described by Section VI, while Section VII presents the paper conclusion and future work.

II. USING GPUS IN COMPONENT-BASED DEVELOPMENT

When developing applications for heterogeneous embedded systems using a *pipe & filter*-type of component model, the developer follows the model specifications to develop software components. The same specifications are used even when the model does not provide directions on how to support GPU within the component. Hence, a component with GPU computations requires encapsulation of all the information and operations needed to support its functionality. For example, the information and operations include choosing the GPU computational resources (e.g., number of threads, grid dimension) to process data or, being GPU unaware, operations to replicate data from and to the main (CPU) memory system.

 ISBN: 978-1-61208-438-1

In the following example, we present a part of a component-based software architecture of a demonstrator that uses a heterogeneous CPU-GPU hardware platform. The demonstrator is an underwater robot developed at Mälardalen University, Sweden. It is used as the running case for the RALF3 research project [6]. The architecture is an abstract view of a component model design that uses a *pipe&filter* interaction style (e.g., ProCom [7], Rubus [4]). The robot has an embedded electronic board containing a GPU, alongside the common CPU. Both processing units have different memory systems. The board is connected to the cameras that are providing a continuous stream of images, and to other sensors and actuators (e.g., pressure sensors, motors).

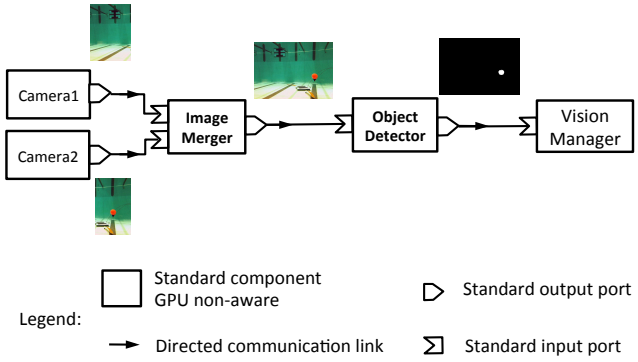

Figure 1. The abstract software architecture of the vision system

Figure 1 presents the robot vision system architecture combined with the data propagation view. The robot uses two cameras which give an extended perspective of the surrounding underwater environment. The physical cameras provide a continuous stream of frames to *Camera1* and *Camera2* software components. The *ImageMerger* receives the frames from the camera components and merge them into a single frame that is filtered by *ObjectDetector*. The filtering process produces a black-and-white frame which eases the identification process of specific objects (e.g., red buoys). The vision system uses the parallel processing power of the GPU hardware for its main data processing activities from *ImageMerger* and *ObjectDetector* components. The black-and-white frame is received by the *VisionManager* component which, based on the position of the objects that have been detected, takes movement decisions for the underwater robot.

Each component, as part of its functionality, accesses particular hardware elements (e.g., CPU, RAM, GPU) in a specific order. The hardware related activities of the vision system are illustrated in Figure 2. The *Camera1* and *Camera2* components, connected to the physical cameras, fetch data frames (two at a time) onto the main (CPU) memory system. The *ImageMerger* component duplicates the two frames onto the GPU memory system and processes them (i.e., merge them into one frame). The component handles inside various operations such as memory allocation, specific GPU-shifting operations and picking suitable GPU computational settings for its processing activity. Having GPU-unaware communication ports, the connection with the *ObjectDetector* component is done in a form that is recognized by the existing component interfaces, i.e., thought the main system. Using the same component model rules, the *ObjectDetector* component has similar actions.

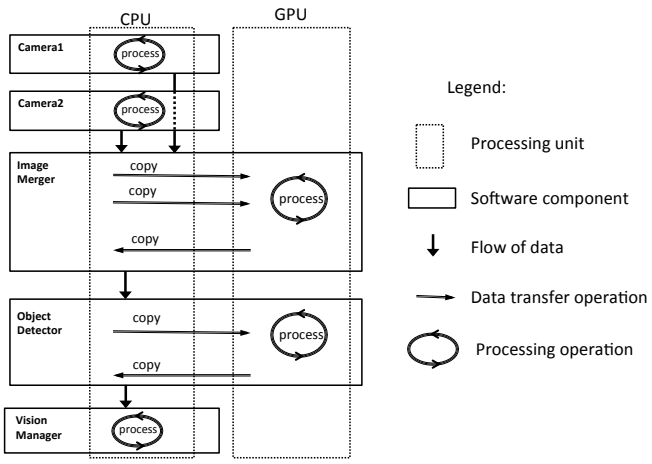

Figure 2. Vision system activities over the hardware

In general, using a *pipe & filter*-type of component model to develop applications for heterogeneous embedded systems has the following disadvantages:

- By being responsible for transferring the data between processing units, each component with GPU capability uses an inefficient copying mechanism as communication method. In most cases, this results in an increased communication over the CPU-GPU hardware bridge (e.g., PCI-Express), which decreases the system performance (e.g., worse reaction time).
- As a side effect of the component encapsulating the same transfer operations, the system contains duplicate code. That is each GPU-based component has copy-from or copy-to CPU operations.
- Each component with GPU capability individually decides (at the development phase) the computational configuration settings. This affects the overall GPU usage of the system and also makes the component less reusable in other contexts.

III. THE GPU-AWARE COMPONENT MODEL EXTENSION

To overcome the drawbacks of using *pipe & filter* component models with no GPU support, we propose a GPU-aware component model extension. In summary, the solution introduces:

- A standardized *configuration interface* through which a component receives GPU computational settings. The assigned settings (limited by the hardware constraints) have a direct impact on the performance of the application. The system, knowing the underlying hardware platform, takes the decision of the computational resources distribution among GPU-aware components. For example, it may distribute the GPU computational resources in such a way that several components can run in parallel (e.g., their summed number of threads should not exceed the total GPU number of threads).
- Dedicated *GPU ports* which are aware of the GPU environment. Instead of communicating using the main

 ISBN: 978-1-61208-438-1

memory, the GPU-aware components communicate directly using the GPU memory.

- Automatically generated *adapters* with dedicated transfer operations. The adapters are automatically introduced when a GPU port is connected to a standard port, in order to facilitate the data transfer operation between the processing units.

According to our proposed solution, the architectural software model of a heterogeneous embedded system is extended in the following way. Ports are classified as GPU ports or standard ports, and the model contains two types of components, GPU-aware and standard components. Standard components can only have standard ports while the GPU-aware components can have both GPU and standard ports. In addition, the software model contains two new inter-component communication elements, the automatically generated communication adapters (CPU-to-GPU and GPU-to-CPU) that resolve the data incompatibility issue between the port types.

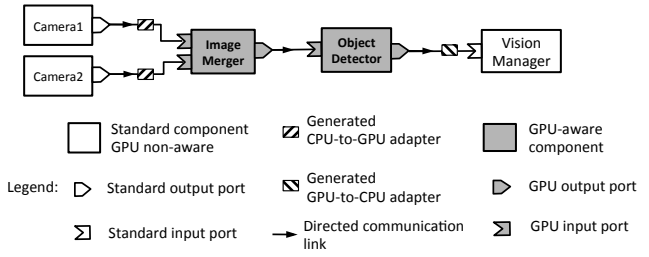

Figure 3. The abstract software architecture of the vision system using a GPU-aware solution

Figure 3 illustrates the abstract software architecture of the vision system using our proposed solution. The system has two GPU-aware components, i.e., *ImageMerger* and *ObjectDetector*, that uses their GPU ports to communicate via the GPU environment. The data provided by the *Camera1* and *Camera2* components are placed onto the GPU by automatically generated CPU-to-GPU adapters. After the GPU-aware components finish their functionality, the output is placed on the main (CPU) memory by another generated adapter (i.e., GPU-to-CPU adapter). This makes the output of the *ObjectDetector* available to the *VisionManager* component.

At startup, the system communicates in a transparent way with the GPU-aware components, connecting to their standardized configuration interfaces. From the architectural software perspective, however, this system-to-component communication mechanism is not graphically represented.

Figure 4 presents the vision system activities of the GPU-adapted software architecture. Compared to the previous non-GPU-aware solution from Figure 2, the newly introduced adapters are handling the data transfer between the CPU and GPU. The two first adapters move data onto the GPU, while the third one transfers the final result back onto the CPU. The *ImageMerger* component, using its GPU ports, takes the frames directly from the GPU (where the adapters placed them) and processes them using the hardware configuration setting received from the system. Also, by having dedicated GPU ports, the communication with *ObjectDetector* is done locally via the GPU memory.

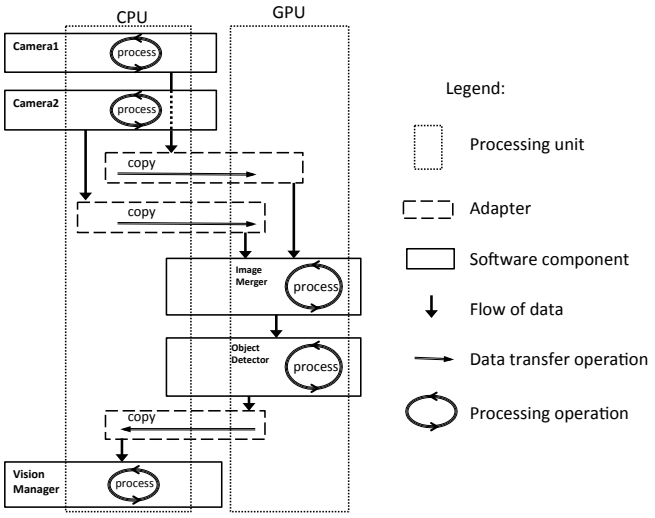

Figure 4. GPU-aware vision system activities over the hardware

The main advantages of our GPU-aware solution are the following:

- Keeping the component communication local on the GPU, whenever possible. This improves the system performance (e.g., better component-to-component communication time) and decreases the communication stress over the hardware CPU-GPU bridge.
- By externalizing the data shifting operations from the component, the component granularity is improved. Also, as a consequence of introducing the specialized adapters, duplicated code is reduced (when the system has at least two GPU-aware components sequentially connected).
- Deciding the GPU configuration of each component at the system level improves the reusability of components. For example, the system can run several components in parallel on the GPU by adjusting their GPU configuration settings, or the same component can be used in different systems with different configuration settings.

IV. EXTENSION IMPLEMENTATION

Next, we give an example of what the extension might look like when implemented. We base the presentation on a simple reference component model to simplify the presentation, and use the vision system from Figure 3 as a running example.

A. An implementation of GPU-aware components

Using C++ as the implementation language, we see a software component as an object with its public member functions describing the component ports and interfaces. Any GPU-aware component is characterized by four member functions, as presented in Figure 5.

The first argument set of the *initialize* function, (i.e., *SIZE_TYPE frame1_in, ...*) specifies the data sizes of the input ports, while the second set (i.e., *SIZE_TYPE frame1_out, ...)* describes the data sizes of the output ports. For GPU-aware components, the *initialize* function has the role of allocating memory on the GPU to hold the produced output

 ISBN: 978-1-61208-438-1

```
class GPU_awareComp{
public:
  void initialize(SIZE_TYPE frame1_in, ...,
                  SIZE_TYPE *frame1_out, ...);
  void config_gpu(int tile, int block_x, int block_y, int block_z);
  void execute(GPU_TYPE in1, ..., GPU_TYPE *out1, ...);
  void free_memory();
};
```

Figure 5. A GPU-aware component implementation details

data. Based on the sizes of the input data, the component uses a GPU API routine (e.g., cudaMalloc) to allocate a specific chunk of memory. The sizes of the allocated memory (i.e., *SIZE_TYPE frame1_out, ...*) will be propagated to the next connected component as input data sizes. The (input and output) arguments are of a structure type and may hold several elements indicating the multi-dimensional aspect of the data, such as three-dimensional matrices or bi-dimensional images. For example, a 2D image argument may be represented by a data structure with two member elements (width and height) that specify the number of image pixels.

Each GPU-aware component receives from the system its GPU execution settings, through the *config_gpu* function. The function parameters describe this settings, such as the three-dimensional size ($block\_x * block\_y * block\_z$) of the thread-blocks unit. A thread-block is a specific unit of thread organization.

The *initialize* and *config_gpu* functions are used once at system startup. After a component allocates the memory to hold its results, it reuses it for all of its executions. The same principle applies for the GPU execution setting; once a component has the GPU setting, it is used every time the component is executed. This design decision is suitable for relatively simple stream processing applications, with static control flow and where one frame is fully processed before starting on the next. Each time a component is invoked, it processes different data using the same GPU setting and reusing the same allocated memory space (considering the dimensions of the streaming frames do not change over time), avoiding the overhead of allocating/deallocating and specifying the execution setting for each frame of the stream.

The *execute* function triggers the core functionality of a component. This function is specified with two sets of arguments. The first set (e.g., *GPU_TYPE in1, ...*), corresponding to its input ports, are pointer variables that specify the GPU locations of the input data. The second set (i.e., *GPU_TYPE *out1, ...*) indicates the GPU locations of the component results, corresponding to the output ports. In case when the component also has standard ports, the types of the ports are used (e.g., TYPE instead of GPU_TYPE).

After the component finishes its executions (it may run several times), it must free the memory that has been allocated to hold its output results. This is done by the *free_memory* function, which uses a GPU API deallocation routine (e.g., cudaFree).

B. Adapters implementation

The GPU-aware components, by communicating directly via the GPU memory, do not have to handle the data shifting activities. Instead, we propose automatically generated software adapters to handle the data transfer between the two computational units.

```
class Cpu2Gpu_adapter{
public:
  void initialize(SIZE_TYPE frame_in);
  void transfer(TYPE in, GPU_TYPE *out);
  void free_memory();
};
```

Figure 6. A CPU-to-GPU adapter implementation details

Figure 6 describes the interface of an adapter that handles the CPU-to-GPU data transfer. Through the *initialize* function, the adapter receives from the system the size of the data to be transferred. Based on this, it allocates memory space on the GPU using a GPU API procedure. The parameter of the *initialize* function, being of a structure type, may hold several elements, which reflects the multi-dimension aspect of the *frame_in*. The *transfer* function uses a GPU API copy procedure (e.g., cudaMemcpy) to transfer data. The first argument represents the CPU location of the input data (from the main memory system), while the second argument holds the GPU location where the data was transferred. The *free_memory* interface deallocates the memory space that was allocated on the GPU.

```
GPU_UCHAR *dev_ptr;

void initialize(SIZE_TYPE frame_in) {
  cudaMalloc(&dev_ptr, 3 * sizeof(dev_ptr) * frame_in.width *
      frame_in.height);
}

void transfer(unsigned char *in, GPU_UCHAR **out) {
  cudaMemcpy(dev_rgb, host_ptr, 3 * sizeof(in) * frame_in.
      width * frame_in.height, cudaMemcpyHostToDevice);
  *out = dev_ptr;
}

void free_memory() {
  cudaFree(dev_ptr);
}
```

Figure 7. An example of CPU-to-GPU adapter implementation using CUDA API

Figure 7 illustrates the implementation details of a CPU-to-GPU adapter using the CUDA API programming model. The adapter transfers a bi-dimensional RGB image (red, green and blue elements of a frame pixel) from the main (CPU) memory to the GPU memory. The input image argument (*img*) is represented by a *SIZE_TYPE* data structure that contains two member elements, *width* and *height*. The *transfer* function uses the main memory frame location specified by the input argument *in*, and executing the *cudaMemcpy* routine, places the image onto the GPU. The memory location of the newly shifted image is memorized by the output argument *out*. In order to make a distinction between the two different memory systems (CPU and GPU), we use two different types to characterize the input and output arguments, i.e., the *unsigned char* and

```
struct SIZE_TYPE {
        int height;
        int width;
};

SIZE_TYPE frame1_in, frame2_in, frame_mrg, frame_filtered;
unsigned char *camera1_in, *camera2_in, *result;
GPU_UCHAR *adp1, *adp2, *merge, *obj;

Camera1.initialize(&frame1_in);
Camera2.initialize(&frame2_in);
Adapter1_CPU2GPU.initialize(frame1_in);
Adapter2_CPU2GPU.initialize(frame2_in);
ImageMerger.initialize(frame1_in, frame2_in, &frame_mrg);
ObjectDetection.initialize(frame_mrg, &frame_filtered);
Adapter3_GPU2CPU.initialize(frame_filtered);
VisionManager.initialize(frame_filtered);

ImageMerger.config_gpu(16, 16, 16, 1);
ObjectDetection.config_gpu(32, 16, 16, 1);

while(stream!=NULL) {
  Camera1.execute(&camera1_in);
  Camera2.execute(&camera2_in);
  Adapter1_CPU2GPU.transfer(camera1_in, &adp1);
  Adapter2_CPU2GPU.transfer(camera2_in, &adp2);
  ImageMerger.execute(adp1, adp2, &merge);
  ObjectDetection.execute(merge, &obj);
  Adapter3_GPU2CPU.transfer(obj, &result);
  VisionManager.execute(result);
}
```

Figure 8. The implementation details of the vision system

GPU_UCHAR types for the main memory and GPU memory location, respectively.

The GPU-to-CPU adapter is implemented in a similar way. The only major difference is located in the *transfer* function, where the first argument describes the GPU location of the image to be transferred, while the other argument describes the main memory (CPU) location of the transferred data.

C. Vision system implementation

We now use the GPU-aware component and adapter implementations previously described to present our implementation of the vision system.

For the robot's vision system, three adapters are automatically generated: two for placing images on the GPU and the other for shifting the final result back on the main (CPU) memory. The initialization of the adapters and GPU-aware components are specified in the upper part of the Figure 8.

The image sizes of the camera output are propagated to the rest of the system according to each component's functionality. For example, the *ImageMerger* component receives the input sizes of the two camera images, and outputs the size of the merged image to the next connected component (*ObjectDetection*). The initialization part is done only once, each adapter and GPU-aware component reusing the same allocated memory to place the continuous stream of frames received from the cameras.

The GPU setting of each GPU-aware component is provided by the system through the *conf_gpu* methods. The system sends once the execution setting to each of the components, which is reused for every image processing activity of the components during the entire application execution. For example, the processing unit of *ObjectDetection* is a block of $16 * 16 * 1$ threads, while is applied on frame tiles of $32 * 32$ pixels.

The execution of the system core functionality is illustrated in the bottom part of the figure, inside the *while* loop. As long as the stream frame flow is not closed (it stops when e.g., the robot mission is completed), the camera components are producing frames which are copied onto the GPU by the CPU-to-GPU adapters. *ImageMerger* uses its input parameter pointers that indicate the GPU memory location of the frames, and outputs, using a GPU-type pointer variable, the location of its result. In the end, the *VisionManager*, using the memory location provided by the GPU-to-CPU shifted data, it processes the data, taking appropriate movement decisions of the underwater robot.

V. Evaluation

To examine the benefits of our proposed solution, we conducted a small experiment to compare the performance with and without the GPU-aware extension, to determine the reduction in communication overhead. To keep it simple, we use only one component, i.e., vertical mirroring of an image, implemented in two variants. A GPU-aware component, developed using our solution, and a standard component developed as described in Section II (encapsulating the data shifting operations between CPU and GPU). We then construct systems of difference sizes by connecting multiple (from 5 to 25) component instances sequentially, using either the GPU-aware or the standard variant.

Two input images are used, one with $1152 * 864$ pixels and a second, larger, with $1152 * 1782$ pixels. The platform on which the experiments were executed consists of an NVIDIA GPU hardware with a Kepler architecture, a 2,6 GHz IntelCore i7 CPU with 16 GB of internal RAM memory. For each case, we executed the system 100 times and calculated the average of the measured times.

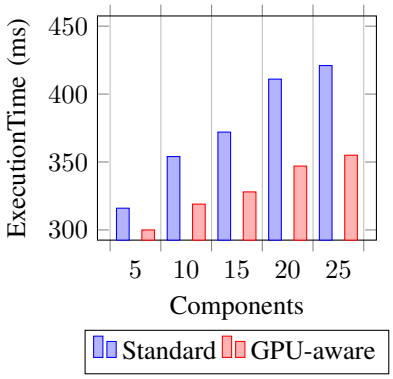

Figure 9. Execution times of two types of systems when processing an image with $1152 * 864$ pixels

Figure 9 illustrates the execution times of the two types of systems while processing an input image of $1152 * 864$ pixels. With standard components, represented in blue color, it takes approximately 315 ms for 5 component to sequentially process

the input image, and 420 ms when the system consists of 25 components. The GPU-aware variant, depicted with red color, need approximately 300 ms to process the same image with 5 components, and 355 ms when the system is composed of 25 components.

The results for the larger input image of $1152*1782$ pixels, presented in Figure 10, show similar improvements.

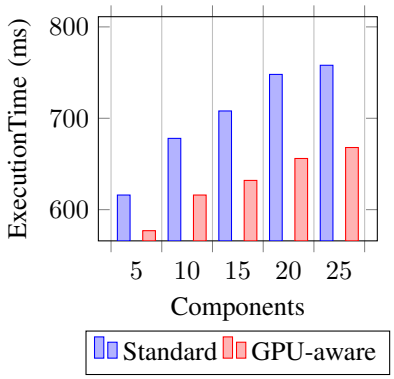

Figure 10. Execution times of two types of systems when processing an image with $1152 * 1782$ pixels

The experiment shows a performance increase of the GPU-aware solution over the standard one where GPU interaction is completely encapsulated in the components. For the smaller input data, the gain is approximately 5% of the total execution time with 5 components, and 16% with 25 components. For the larger input, the gain is 6% and 12% with 5 and 25 components, respectively.

VI. Related Work

There are several component-based approaches that in different ways target GPU-based systems, as discussed in the following paragraphs.

Elastic Computing [8] is a framework that, based on a library that contains pre-built "elastic functions" for specific computations, determines offline the optimized execution configuration for a given platform. The framework does not manage the execution of GPU devices, which is done internally inside the elastic functions, alongside with the resource allocation and data management.

Kicherer et al. [9] use a component-based approach to propose a performance model suitable for on-line learning systems. The disadvantage of their approach is that the data management does not handle transfer operations for the GPU execution. Hence, the data transfer between the main memory and the GPU device is done internally by the library of the performance model. Differing from both of the presented approaches, the data management and resource allocation is done automatically (by adapters), from outside of the component level.

A theoretical component model is proposed by Stoinski [10] to support data stream applications by adding a dedicated port which enables data stream communication (e.g., MPEG1 video) between components. Comparable to this theoretical approach, we are extending the hardware platform specification to include GPUs, and enriching the component interfaces to enable data (e.g., stream of frames) communication between components via the GPU memory system.

The PEPPHER component model [11] constructs an environment for annotations of C/C++ components for heterogeneous systems, including (multi-)GPU based systems. The model provides different (sequential or parallel) implementation variants (e.g., one for multi-core CPU and another for GPU) for the same computational functionality (component), together with the meta-data (tunable parameters). The composition code of the component is in the form of stubs (proxy or wrapper functions). In addition to this work, we address, transparently, the system-to-component communication for the GPU execution settings. The memory management issue is handled by smart containers. Contrasting their approach, we use automatically generated adapters which can be seen as a high level memory management elements.

Regarding code generation, there is much work done in automatically porting sequential (or parallel) CPU source code for GPU execution. Several programming languages have such GPU translators, such as Java [12], C [13], C++ [14], OpenMP [15], Python [16] or Matlab [17]. For our work, these approaches can be used to generate parts of the implementation of GPU-aware components.

VII. Conclusion

Despite the growing trend of using heterogeneous platforms for embedded systems, there is a lack of efficient ways to address the CPU-GPU combination in the existing *pipe & filter* component models. When a component model does not provide dedicated means to specifically handle GPUs, each component have to redundantly encapsulate the same GPU specific operations and settings required to meet and support their functionality. Our solution tackles the inefficient development by proposing a GPU-aware component model extension. With our method, the components are aware of the GPU environment by having specialized GPU interfaces and ports which facilitates the component communication via the GPU environment.

The benefits of our solution include:

- The system performance is increased from reducing the component communication overhead and keeping data locally on the GPU when possible, as indicated by the experiment in Section V.
- Improved component granularity and reduced code duplication, as a consequence of introducing specialized generated adapters for data shifting operations.
- An increased reusability of components by adjusting the components GPU configuration setting at the system level. For example, the system can run several components in parallel on the GPU by adjusting their GPU configuration settings, or the same component can be used in different systems with different settings.

As future work we want to increase the flexibility of the GPU memory management to support also more dynamic memory allocation. Moreover, our work may be extended by supporting parallel execution of GPU-aware components. Another possible thread of future work includes implementing the method in some existing component model, e.g., Rubus [4] or ProCom [7].

 ISBN: 978-1-61208-438-1

ACKNOWLEDGMENT

Our research is supported by the RALF3 project [18] through the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] D. Geronimo, A. M. Lopez, A. D. Sappa, and T. Graf, "Survey of pedestrian detection for advanced driver assistance systems," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 32, no. 7, 2010, pp. 1239–1258.

[2] P. Michel et al., "GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing," in Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on. IEEE, 2007, pp. 463–469.

[3] AUTOSAR Development Partnership, "AUTOSAR Technical Overview, v4.2," http://www.autosar.org, (accessed June 28, 2015).

[4] Arcticus Systems, "Rubus Component Model," https://www.arcticus-systems.com, (accessed June 28, 2015).

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

[6] C. Ahlberg et al., "The Black Pearl: An autonomous underwater vehicle," Mälardalen University, Tech. Rep., June 2013, published as part of the AUVSI Foundation and ONR's 16th International RoboSub Competition, San Diego, CA.

[7] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnkovic, "A component model for control-intensive distributed embedded systems," in Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008). Springer Berlin, October 2008, pp. 310–317.

[8] J. R. Wernsing and G. Stitt, "Elastic computing: A portable optimization framework for hybrid computers," Parallel Computing, vol. 38, no. 8, 2012, pp. 438–464.

[9] M. Kicherer, F. Nowak, R. Buchty, and W. Karl, "Seamlessly portable applications: Managing the diversity of modern heterogeneous systems," ACM Transactions on Architecture and Code Optimization (TACO), vol. 8, no. 4, 2012, p. 42.

[10] F. Stoinski, "Towards a component model for universal data streams," Eighth IEEE International Symposium on Computers and Communication, 2003, 2003.

[11] 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.

[12] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in Euro-Par 2009 Parallel Processing. Springer, 2009, pp. 887–899.

[13] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in Compiler Construction. Springer, 2010, pp. 244–263.

[14] F. Jacob, J. Gray, Y. Sun, and P. Bangalore, "A platform-independent tool for modeling parallel programs," in Proceedings of the 49th Annual Southeast Regional Conference. ACM, 2011, pp. 138–143.

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

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," Parallel Computing, vol. 38, no. 3, 2012, pp. 157–174.

[17] A. R. Brodtkorb, "The graphics processor as a mathematical coprocessor in MATLAB," in Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on. IEEE, 2008, pp. 822–827.

[18] RALF3, "Software for Embedded High Performance Architecture," http://www.mrtc.mdh.se/projects/ralf3/, (accessed September 10, 2015).

 ISBN: 978-1-61208-438-1