The realization of flexible GPU components in Rubus

Gabriel Campeanu
Bombardier Transportation
Sweden

Jan Carlson, Séverine Sentilles
Mälardalen University
Sweden

June, 2019
revised October, 2019

Abstract

This technical report collects details on the realization of flexible GPU components in Rubus, including *i)* the automatically generated code of adapters, *ii)* the generated code implementing a component group, and *iii)* the generic API providing transparent access to multiple platforms and OpenCL versions.

1 Introduction and overview

In previous publications [1, 2, 3, 4], we have addressed the lack of support for GPU development in component models targeting embedded systems, and proposed a solution based on the concepts of *flexible components*, *component groups* and *adapters*. This technical report collects details on the realization of these concepts in the Rubus component model [5]. For a thorough description of the overall approach and the proposed concepts, see [4].

In the code listings, bold text in angular brackets represent expressions that are evaluated/expanded as part of the code generation. For example ⟨**`p_out.name`**⟩ would be replaced by the name of the port p_{out}.

The report is organized as follows: Section 2 presents the automatically generated adapter code. Section 3 describes the code that is automatically generated to define the Rubus component realization of a component group. Finally, Section 4 presents the generic API providing transparent access to multiple platforms and OpenCL versions.

2 Automatically generated adapter code

In order to make flexible components platform independent while avoiding unnecessary data copying between CPU and GPU memory, *adapters* are automat-

Listing 1: Constructor code of adapters

```
1  /* create memory buffers for the (one) output port */
2  void *result_adp = apiCreateBuffer(settings->contex, CL_MEM_WRITE_ONLY,
        <p_in.width*p_in.height*p_in.size>,NULL,NULL);
3
4  /* connect the output port to the created buffers */
5  <p_out.name>->data = (unsigned char*) result_adp;
```

Listing 2: The adapter behavior function

```
clEnqueueWriteBuffer(settings->cmd_queue, result_adp, CL_TRUE, 0,
     <p_in.width*p_in.height*p_in.size>, <p_in.name>->data, 0, NULL, NULL);
```

ically generated where needed (i.e., between components with different allocations) to manage the needed data copying. This section describes the generation of the adapters' constituent parts, i.e., constructor, behavior function and destructor.

The constructor. The adapter has one input data port *p_in* and one output data port *p_out*, both of reference type. The adapter's constructor, presented in Listing 1, allocates memory (line 2) corresponding to the size of input data, on the appropriate location, i.e., the device (for *inAdapter* adapters) or main memory (for *outAdapter* adapters). The output port is linked to the location that holds the copied input data (line 5).

Behavior function. The generated code of this part handles the transfer of data to or from the GPU memory, according to the allocation of the connected components. The *clEnqueueWriteBuffer* is synchronous (i.e., returns the control after it finishes) due to the usage of *CL_TRUE* flag.

The destructor. Opposite to the adapter's constructor that allocates memory space for the input data, the adapter's destructor releases this memory.

3 Realization of component groups

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

Listing 3: Destructor code of adapters

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

Listing 4: Interface code

```
1  /* device-settings for each flexible component C */
2  <counter = 1>
3  <foreach $C$ in $G$>
4  typedef struct {
5         int blockDim_x;
6         int blockDim_y;
7         int gridDim_x;
8         int gridDim_y;
9         cl_context context;
10        cl_command_queue cmd_queue;
11        cl_device_id device_id;
12  }settings<counter+=1>;
13 <endforeach>
14
15 /* the group input ports */
16 <counter = 1>
17 typedef struct {
18 <foreach p in $I_{\text{in}}^{G}$>
19        <p.type> *<p.name>;
20 <endforeach>
21 <foreach $C$ in $G$>
22        settings<counter> *cfg<counter>;
23        <counter += 1>
24 <endforeach>
25 }IP_SWC_iArgs;
26
27 /* the group output ports */
28 typedef struct {
29 <foreach p in $I_{\text{out}}^{G}$>
30     <p.type> <p.name>;
31 <endforeach>
32 }OP_SWC_iArgs;
33
34 /* the interface of the group */
35 typedef struct {
36   IP_SWC_iArgs IP;
37   OP_SWC_iArgs *OP;
38 }SWC_Group_iArgs;
```

grouped components. The destructor releases the group allocated resources.

3.1 The interface

The interface is composed of two parts, the input and the output. In Listing 4, the *SWC_Group_iArgs* interface is defined as a structure (lines 35-38) with two elements corresponding to the output and input interfaces. The output interface *OP_SWC_iArgs* is constructed as a structure (lines 28-32), where the elements are the data ports of the group output interface I_{out}^{G}. Similarly, *IP_SWC_iArgs* is a structure that encloses the data ports of the group input interface I_{in}^{G}.

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

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

3.2 The constructor

The *constructor*, illustrated in Listing 5, encloses all the information regarding the group initialization, as follows. The listing starts by allocating memory for each flexible component from the group. That data received by a component through the input ports is the input data for the functionality, while the functionality outcomes are sent through the output ports. Therefore, corresponding to each output data port, we allocate memory to hold the functionality results. Due to the specifics of OpenCL, a kernel function must store a simple output value (e.g., integer value) in a one-value memory buffer. Thus, we allocate one-value memory buffers for simple output ports (line 4). For data at reference ports, the memory buffer is allocated with an appropriate size (line 8). Moreover, in line 15, the reference ports that are considered output ports of the group are linked to the corresponding memory locations. This is done because these ports will be wired to outside-of-the-group ports, and the system communication will be accomplished by using the values of the connected ports.

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

3.3 The behavior function

The execution of the group functionality is enclosed in the *behavior function* (Listing 6) which is performed every time the group is activated. To execute the functionality using the OpenCL model, the host needs to send to the selected

Listing 5: Constructor code

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

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

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

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

<$F^{C}$>
}";
<counter_kernel += 1>
<endforeach>

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

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

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

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

Listing 6: Behavior function

```
/*Set the kernel arguments of each enclosed component*/
<counter_kernel = 1>
<counter_arg = 0>
<foreach $C$ in $G$>
 <for each $p$ in $I^{C}_{\text{sim\_in}}$>
  /* for simple input ports of flexible components that are considered input ports of the group */
  <if (p in $I^{G}_{\text{sim\_in}}$)>
    apiSetKernelArg(kernel<counter_kernel>,<counter_arg+=1>, sizeof(<p.type>), (void*)&<p.name>);
  /* for simple input ports of flexible components that are not input ports of the group */
  <else>
    apiSetKernelArg(kernel<counter_kernel>,<counter_arg+=1>, sizeof(<p.type>), (void*)&result_<(p.connect).name>);
  <endif>
   <counter_kernel+= 1>
 <endforeach>
 <for each $p$ in $I^{C}_{\text{sim\_out}}$>
   apiSetKernelArg(kernel<counter_kernel>,<counter_arg>, sizeof(<p.type>), (void*)&result_<p.name>);
   <counter+= 1>
 <endforeach>
 <foreach $p$ in $I^{C}_{\text{ref\_in}}$>
 /* reference input ports of flexible components that are input ports of the group */
  <if (p in $I^{G}_{\text{sim\_in}}$)>
    apiSetKernelArg(kernel<counter_kernel>,<counter_arg>, <p.width*p.height*p.size>, (void*)&<p.name>);
  /* reference input ports of flexible components that are not input ports of the group */
  <else>
    apiSetKernelArg(kernel<counter_kernel>,<counter_arg>, <p.width*p.height*p.size>, (void*)&result_<(p.connect).name>);
  <endif>

   <counter+= 1>
 <endforeach>
 <for each $p$ in $I^{C}_{\text{ref\_out}}$>
   apiSetKernelArg(kernel<counter_kernel>,<counter_arg>, <p.width*p.height*p.size>, (void*)&result_<p.name>);
 <endforeach>
 <counter_kernel+=1>
<endforeach>

/* Execute the OpenCL kernels of the flexible components */
<counter=1>
<foreach $C$ in $G$>
   clEnqueueNDRangeKernel(settings->cmd_queue, kernel<counter>, 2, NULL, total_thrd<counter>, group_thrd<counter>, 0, NULL, NULL);
   <counter+=1>
<endforeach>

/* copy the simple output(s) to the corresponding simple output port(s) of the group */
<foreach $C$ in $G$>
 <foreach p in $I^{G}_{\text{sim\_out}}$>
   apiEnqueueReadBuffer(settings->cmd_queue, result_<p.name>, CL_TRUE, 0, sizeof(<p.type>), &<p.name>, 0, NULL, NULL);
 <endforeach>
<endforeach>
```

Listing 7: Destructor code

```
1   /* Clean up */
2   <counter_kernel = 1>
3   <foreach $C$ in $G$>
4     clReleaseKernel(kernel<counter_kernel>);
5     <counter_kernel+=1>
6   <endforeach>
7     clReleaseProgram(program);
8   <foreach $C$ in $G$>
9    <foreach p in $I_{\text{out}}^{C}$>
10     apiReleaseBuffer(result_<p.Name>);
11   <endforeach>
12  <endforeach>
```

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

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

3.4 The destructor

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

Listing 8: The *apiCreateBuffer* function

```
void *apiCreateBuffer(cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)
{ // Create memory buffer on the device
#if !defined(CL_VERSION_2_0) && ( defined(CL_VERSION_1_2) || defined(CL_VERSION_1_1) )
       //Distinct memory allocation buffer";
       return ((void *)clCreateBuffer(context, flags, size, host_ptr, errcode_ret));
#endif

#if defined(CL_VERSION_2_0) && defined(CL_VERSION_2_1)
       //shared virtual memory
       cl_device_svm_capabilities caps;

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

Listing 9: The *apiSetKernelArg* function

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

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

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

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

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

4 Implementation of generic API functions

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

- the *apiCreateBuffer* function allocates GPU memory buffers,
- the *apiReleaseBuffer* function deallocates existing GPU memory buffers,
- the *apiTransferBuffer* function transfers data between different memory systems, and
- *apiSetKernelArg* to set up the parameters for GPU functions.

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

The *apiSetKernelArg* function, presented in Listing 9, abstracts the different functions that are required to set the kernel arguments, on different type of GPU architectures. For example, while for platforms with distinct memory systems, the *clSetKernelArg* function is needed, on platforms with full shared memory, *clSetKernelArgSVMPointer* is required.

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

References

[1] G. Campeanu, J. Carlson, S. Sentilles, 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.

[2] G. Campeanu, J. Carlson, S. Sentilles, Developing CPU–GPU embedded systems using platform-agnostic components, in: 43rd Euromicro Conference on Software Engineering and Advanced Applications, 2017.

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

[4] G. Campeanu, GPU support for component-based development of embedded systems, Ph.D. thesis, Mälardalen University, Sweden (2018).

[5] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, K.-L. Lundbäck, The Rubus component model for resource constrained real-time systems, in: Industrial Embedded Systems, 2008. SIES 2008. International Symposium on, IEEE, 2008, pp. 177–183.