CDSC CnC-HC Documentation

Alina Sbîrlea
Rice University
Houston, TX 77005, USA
Email: alina@rice.edu

I. A CLOSER LOOK INTO THE MEDICAL IMAGING PIPELINE

In this section we will go over the CnC description of the medical imaging pipeline and explain how this application was modelled in the CnC graph language. Listing 1 shows the resulting CnC graph.

```
1  < int [1] emtv_tag >;
2  < int [1] denoise_tag > ;
3  < int [1] reg_tag > ;
4  < int [1] seg_tag > ;
5
6  [ float* em_output ] ;
7  [ float* emtv_output ] ;
8  [ float* denoise_output ] ;
9  [ float* registration_output ] ;
10 [ float* final_output ] ;
11
12 <emtv_tag>    :: ( em            @FPGA=1 );
13 <emtv_tag>    :: ( tv            @CPU=1 );
14 <denoise_tag> :: ( denoise       @CPU=3, GPU=33 );
15 <reg_tag>     :: ( registration  @FPGA=3, GPU=1 );
16 <seg_tag>     :: ( segmentation  @GPU=9 );
17
18 [emtv_output : p-1]       -> (em : p)              -> [em_output : p];
19 [em_output : p]           -> (tv : p)              -> [emtv_output : p];
20 [emtv_output : s]         -> (denoise : s)         -> [denoise_output : s];
21 [denoise_output : 5]      -> (registration : k)    -> [registration_output : k];
22 [registration_output : k] -> (segmentation : k)    -> [final_output : k];
23
24 env -> [emtv_output : 0], <emtv_tag : {1 .. 6}>, <denoise_tag : 5>;
25 env -> <reg_tag : {0 .. 10}>, <seg_tag : {0 .. 10}>;
26 env <- [final_output : {0 .. 10}];
```

Listing 1: CDSC Pipeline described in CnC graph form

Let us first remember the CnC constructs. CnC graphs have three components: item collections, control collections and step collections. Item collections store data in the form of $< key, value >$ pairs, and these items are dynamic single assignment. Control collections are sets of tags and are each associated with one or more step collections. This means that adding a tag to a control collection will spawn one or more steps with that tag. Step collections group together computational kernels with the same functionality and parametrized with a tag.

In the pipeline example in Listing 1, lines 1-4 contain control collection declarations. For example, line 1 declares a control collection named $emtv\_tag$, which contains tags which are single dimensional tuples with type int. Thus, any int value is a valid tag for control collection $emtv\_tag$. In general, tags may be multidimensional, such as having a pair or a triplet, but each of their components has the type int.

The next five declarations (lines 6-10) are for item collections. Each line declares a collection of type $float*$, which means that each value in this collection has the type $float*$.

Lines 12-16 are step prescriptions, which means they define which control collection will start which steps. In this example, lines 12 and 13 define a control collection ($emtv\_tag$) which prescribes two steps: em and tv. The next lines (14-16) define three other prescription relation, each for a different step. The CnC semantics impose that a step be prescribed by only one control collection. On each prescription relation, a step may specify its affinity with one or more devices. For example, on line 12, step em is defined to have affinity equal to 1 with the FPGA. This implies that step em cannot run anywhere else except on the FPGA. The user will thus provide the appropriate code for running the em computation on an FPGA, while the CnC runtime will handle the assignment of the step to the appropriate device. On line 14, step $denoise$ is defined to have affinities with the CPU and the GPU, with a higher affinity with the GPU. In this situation, the CnC runtime will favor running $denoise$ on the GPU if the device is found to be available and there are no requests with higher affinities to use the device. Apart from

the affinity information, the user should provide a version of the $denoise$ step for the CPU and for the GPU, then rely on the CnC runtime to provide the best overall application performance based on the machine load and the step affinity values. If no affinities are defined for a step, it is assumed the step can only run on the CPU.

Lines 18-22 describe input-output relations for each step. For example, on line 18, step em with some tag p is defined to read from item collection $emtv\_output$ an item with the key=p-1, and to write into item collection em_output an item with key=p.

Lines 24-26 describe input-ouput relations for the environment. The writes from the environment are performed in order to provide the initial data and start the graph execution, while the reads from the environment are performed in order to get the results of the graph computation once this has finished. For example, line 24 specifies that the environment writes an item with key=0 into item collection $emtv\_output$, a range of tags with values from 1 to 6 into the control collection $emtv\_tag$ (thus starting 5 instances of each of the em and tv steps) and a single tag with value=5 into the control collection $denoise\_tag$ (thus starting the $denoise$ step with tag 5). Similarly, the environment starts more steps on line 25. On line 26, the environment is defined to read from item collection $final\_output$, ten item with tags ranging from 0 to 10 (exclusive), once the execution of the graph has terminated.

II. CnC-HC translator

The CnC-HC distribution offers the CnC-HC runtime and the CnC-HC translator. A user need not concern about the runtime as that will be used implicitly in his application, but he will have to call the translator explicitly on the graph file to obtain a series of helper files. Calling the translator is done with the command:

$cncc\_t\ graph\_file\_name.cnc$

Calling the translator will generate a series of files which we shall detail here. The need for generation of "glue code" is due to application dependent components like item collections and steps but also to the usage of the C language. Some of the artefacts that we need to generate include: an unique identifier for each step, methods that initialize or delete a CnC graph, methods that launch the execution of other steps and methods that link the user code to the CnC runtime.

In the following files: Common.* Context.* Dispatch.* the translator generates all the code necessary to create the link between the user code and the CnC runtime. They are normally not modified, and are overwritten on subsequent calls of the translator. If tag functions are not present in the graph file, then the Common.* files need to be edited to call the appropriate Get functions. For using user-defined types or global constants, the user may add the proper includes in Context.h. Since the user should not be concerned or modify the contents of Context.* Dispatch.*, we do not present code snippets for those files here. We present in Listing 2 a simplified version of the code generated by the translator for step em in Common.hc. This should clarify how data is read in the generated code and further passed as parameter to the user step code.

```
1 void* em_gets(char * tag, Context * context, Step* step){
2         int p = getTag(tag, 0);
3         float* emtv_output0;
4         char* tagemtv_output0 = createTag(1, p-1);
5         CNC_GET((void**) & (emtv_output0), tagemtv_output0, context->emtv_output, step);
6
7         em( p, emtv_output0, context );
8         return 0;
9 }
```

Listing 2: Simplified Get calls for step em

On line 2, the first component of the step tag is obtained in a local variable p. On line 3, the code declares a variable to hold the data that is about to be read. On line 4, a runtime method is invoked to create a new tag having one component with value p-1. On line 5, the code calls the CnC runtime function CNC_GET and retrieves the item with tag p-1 from collection $emtv\_output$. This call may abort if the data is not present and the runtime will handle the restart when the data becomes available (this behaviour is offered by the Rollback and Replay runtime). The value obtained, together with the tag p are passed as parameters to the actual step code on line 7.

We present in Listing 3 the actual code generated for the Get calls for step em. Due to the fact that CnC-HC provides two alternate runtimes, the actual generated code contains an #ifdef which provides the alternate behaviour. We described in Listing 2 the behaviour of the Rollback and Replay runtime. The alternative is the Data Driven runtime, when the data is certainly available when the em_gets is called. As such, instead of calling CNC_GET, the code calls a runtime method to obtain the data stored specifically for this step. For this magic to work, there is an additional function provided ($em\_dependencies$) which details the inputs needed by the step in a manner very similar to the one we described above.

We have thus far presented the generated code that the translator overwrites at every call, and which, if the user provides accurate tag functions, need never be edited or read.

The translator also generates step stubs, a main method, and a Makefile.

```
void* em_gets(char * tag, Context * context, Step* step){

        int p = getTag(tag, 0);
        float* emtv_output0;
#ifndef _DATA_DRIVEN
        char* tagemtv_output0 = createTag(1, p-1);
        CNC_GET((void**) & (emtv_output0), tagemtv_output0, context->emtv_output, step);
#else
        getNextData((void**) & (emtv_output0), step);
#endif
        em( p, emtv_output0, context );
        return 0;
}

void em_dependencies(char * tag, Context * context, Step* step){
        int p = getTag(tag, 0);
        char* tagemtv_output0 = createTag(1, p-1);
        addDependency( tagemtv_output0, context->emtv_output, step);
        addDependencyFinished(step);
}
```

Listing 3: Actual generated Get calls for step em

A step stub will be generated for every step defined in the graph. These need to be edited by the user with the afferent computation. The step files will not be overwritten on subsequent calls of the translator. The role of the stubs is to provide suggested code for Put and prescribe calls based on the information retrieved from the .cnc file. These calls are optional, meaning that, for a particular step instance, the user may choose not to Put (write) all data declared in the graph file or not to prescribe (spawn) all steps specified. The $full-auto$ option makes Puts and prescribes compulsory thus avoiding the need for edits in the stubs, but restricting the application space.

The Main.hc file contains the main() for the program and has generated code on how a graph should be created and places initialized. It is not overwritten on further translator calls. As with step-stubs, using the $full-auto$ option is designed to help the user avoid edits. The user does however still need to provide the implementations for the computational steps(these can be in their own files, added to the ones generated by the translator; the method names will have the format stepname_device and the signature can be found in Dispatch.h)

The Makefile file contains a reference Makefile for building the application using the generated files. It is not overwritten on further translator calls.

III. Changing a CnC description

In this section we look into how a CnC description can be changed to add, remove or modify steps and how the generated files change in each situation

A. Adding and removing elements in the graph

In order to add step, item or control collection, the graph file is simply edited to include the new information. Let us assume that we want an additional image analysis step to follow the segmentation. The following lines then be added to the .cnc file:

```
//Add a new control collection for spawning the analysis steps.
< int [1] analysis_tag>;
//Add a new item collection for storing the segmentation output
[ float* segmentation_output ]  ;
//Write the prescription rule indicating that analysis_tag prescribes analysis steps.
<analysis_tag>  :: ( analysis );
//Modify the I/O rule for segmentation to write into segmentation_output collection
[registration_output : k] -> (segmentation : k)    -> [ segmentation_output : k ];
//Add a new I/O rule for the analysis step
[segmentation_output : k] -> (analysis : k)    -> [ final_output : k ];
//Add that a set of control tags is written into analysis_tag from the environment
env ->  <analysis_tag : {0 .. 10}>;
```

Listing 4: Adding a step to pipeline.cnc

Similarly removing elements from the graph involves simply removing their declaration and presence in prescription and I/O rules.

Rerunning the translator after adding or removing an element will have the following result:

1) Adding an item collection results in the generation of a declaration for that collection in the cnc graph internals. Users will be able to refer to it in the step code when doing a Put into that collection by using the context (e.g., context-¿segmentation_output).
2) Adding a step collection results in a stub being generated for that step. Existing stubs will not be overwritten. Additional internal data is generated for that step, which is transparent for the user (e.g. a unique identifier).
3) Removing an item declaration results in its removal from the context.
4) Removing a step removes the internal data for that step, but does not remove the stub. The translator will not delete files. It is the user's responsibility to also update the Makefile, the translator will not overwrite it if the file exists.

B. Updating the graph

Apart from adding or removing item, step or control collection from the graph, it may be that the user decides to change his application by modifying the I/O relations or by changing the tag space for a step/item. Here are a few examples and their impact of the generated code.

All changes made to the outputs of a step translate in a change in the step stub. However, since existing stubs are not overwritten, these changes will not update the stubs with the new information. The stub can be removed (then the translator will regenerate one with the latest information, or the user can update the information himself.

All changes made to the inputs of a step translate in an update of the generated Get calls found in Common.hc. They also may change the step signature which is only updated in the file containing the function definition (Common.h) but not in the step stub (again, these are not overwritten or modified in any way if they exist) Thus, if the number of inputs changes or if a step is changed to read items from other collections, of different types etc, then the user needs to update the step signature in the generated step stub.

If the tag space for a step changes (e.g. step em is changed to be identified by 2 components p and q), the signature of the step needs to be updated to include the change. As with inputs, the only place where the translator does not update the change is the step stub.