CnC-OCR Tutorial

Presented by Nick Vrvilo (Rice University).
4th Traleika Glacier Applications Workshop. April 6, 2015.

Setup

Navigate to the hll/cnc directory in your copy of the X-Stack repository, and run this command:

```
source setup_env.sh
```

To test if the setup was successful, try running the CnC translator utility:

```
cncocr_t
```

If everything is set up correctly, the translator should report a usage (help) message. Source code for this tutorial is located in the hll/cnc/examples/tutorial directory.

App design sketch

In this tutorial we will build a 1-dimensional, 3-point stencil application using the CnC-OCR framework. As with any program, we should start by sketching the high-level design for our program before we jump into any coding.

We will define our 3-point stencil over a vector of size *N* with the following equations:

$$a_{0,t} = a_{N-1,t} = 1$$

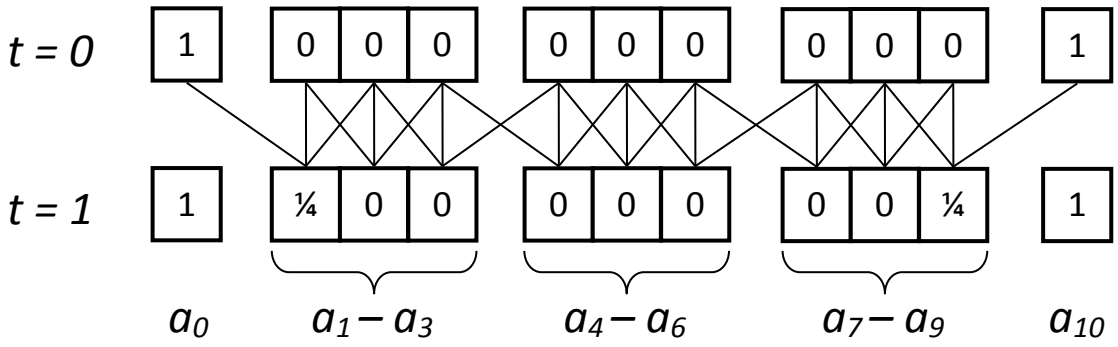
$$a_{i,0} = 0$$

$$a_{i,t} = \frac{1}{2} a_{i,t-1} + \frac{1}{4}\left(a_{i-1,t-1} + a_{i-1,t+1}\right)$$

where $a_{i,t}$ is the value of the vector at index *i* on time-step *t*. The precedence of the equations for computing the vector values should follow the order they given above. Given that the vector's first and last entries are fixed to the value 1 on all time-steps, we expect the vector to eventually converge to a vector of all 1's. We also expect the average of the vector's values to converge to 1.

Since we are designing the application with parallelism in mind, we should assume the stencil operations will work on contiguous chunks of the vector (tiles), concurrently computing multiple tile updates. The first and last vector elements are constant-valued, we can simply assume that they always have the value 1; therefore, we will exclude the first and last entries from our tiles.

It would be helpful to sketch out the first time-step of the stencil computation. Let us assume that we have a vector of length 11. By excluding the first and last elements, we can chunk the remaining entries into 3 tiles, each containing 3 entries. The following figure illustrates the first time-step's computation:



Aside from the first and last entries in a tile, all values read for when computing the updated values are contained within that tile. Each tile needs the edge values from its left and right neighbors. In other words, we need a boundary exchange on the left and right edge of each tile to get the left and right dependences. However, the first tile gets the constant value 1 for its left dependence, and the rightmost tile gets 1 for its right dependence. This process repeats on each successive time-step.

Building the CnC graph

You can use the empty graph specification file provided in the 0-blank.slate directory as a starting point for this project.

The CnC programming model is built around graph-based computation. A CnC computation graph is composed of data nodes, computation nodes, and dependence edges.

The CnC-OCR graph translator tool takes a textual graph specification as input, and produces a skeleton OCR project as output. We will write the graph specification file incrementally as we build our graph. We will save our graph specification in a file called Stencil1D.cnc.

Item collections

First, we should identify the data in our program, which will comprise our item collections. As described above, to compute an updated tile, we need that tile's current values, plus the border elements to the left and right of the current tile. We could have a single item collection, consisting of all the tiles in our computation. To get the left and right dependences, we would get the neighboring tiles, and read the values of the first or last entry in the tile. However, in a distributed system this might require shipping an entire tile to a remote host in order to read a single element. Instead, we'll declare three item collections: one for our tiles, one for left-hand dependences, and one for right-hand dependences. Assuming each vector entry is of type *float*, we would add the following to our graph spec file:

```
// item collections
```

```
[ float tile[TILE_SIZE] ];
[ float *fromLeft  ];
[ float *fromRight ];
```

CnC item nodes are represented graphically using rectangular nodes. Similarly, we use square brackets to denote an item collection in the textual graph representation. Since our tiles contain multiple *float* values, we declare it using array syntax. Since the left and right dependences only consist of a single *float* value each, we declare them with pointer syntax. The array and pointer declarations produce equivalent types in the generated OCR code; however, the generated skeleton code differs for the two cases since the former is assumed to be a *vector* of values, whereas the latter is assumed to be a single scalar value. Note that whitespace is not significant in the graph spec language, so we can add extra spaces to align the closing brackets to make things more symmetric. The graph spec language support C-style block comments /* … */ and C++-style line comments //….

The array syntax for declaring an item collection also takes an optional size. If you do not include a size, the generated code will give a size of 1, and add a /* TODO */ comment to remind you to update the size to something more meaningful. However, it is probably better to specify the size within the graph spec. We will assume that TILE_SIZE is declared with the number of elements in a tile.

Step collections

Next we need to identify the discrete computation steps (tasks) that our app will perform. For our stencil, the update operation on each tile seems like a good granularity for the computation steps. In addition, every CnC program has a pseudo-step to initialize the graph, and another to do something with the results of the computation steps. We'll add these three steps/pseudo-steps to the graph spec:

```
// step collections
( $initialize );
( stencil     );
( $finalize   );
```

We named our computation step *stencil* since it does a stencil update. The pseudo-steps have the special names *$initialize* and *$finalize*. The dollar sign denotes special keywords in the graph spec file that will be translated to some other name or representation in the generated source code. This helps differentiate built-in names from names chosen by the user for collection names and other identifiers.

Step tags and item keys

In CnC, individual step instances are distinguished by a unique *tag*, and item instances are similarly distinguished by a unique *key*. For simplicity we sometimes less precisely refer to tags and keys both as tags. Tags and keys are restricted to be integer tuples in CnC-OCR, although the general CnC model allows for more general types. In our application, all steps and items can be uniquely identified by the 2-tuple index, time-step (*i, t*). We should add the keys/tags to the graph spec:

```
// item collections
[ float tile[TILE_SIZE]: i, t ];
[ float *fromLeft:  i, t ];
```

```
[ float *fromRight: i, t ];

// step collections
( $initialize: () );
( stencil: i, t );
( $finalize: () );
```

Note that the initializer and finalizer pseudo-steps do not need a tag since we only need one of each; thus, these are given empty tags, signified by a pair of empty parentheses *()*.

Data I/O

Now that we have all of the graph nodes, we can start adding some edges. Edges represent inputs to a step (i.e. dependences) and outputs from a step. Steps can only depend on item instances for input, but they can have step and/or item instances as output. We'll just look at the data for now.

We can start with the *stencil* step, since that is where all the work happens. Each stencil step takes as input the tile's values, plus the right and left neighbors' border elements, all from the previous time-step. It produces an updated tile, plus its own border elements (which will be read by neighbor stencil steps), all for the current time-step. We update the stencil step function as follows:

```
( stencil: i, t )
 <- [ tile: i, t   ], [ fromLeft: i,   t   ], [ fromRight: i,   t   ]
 -> [ tile: i, t+1 ], [ fromLeft: i+1, t+1 ], [ fromRight: i-1, t+1 ];
```

The step inputs follow the inward-facing <- arrow, and the outputs follow the outward-facing -> arrow. Note that all of the expressions used for the referenced item instances' keys are defined in terms of the variables composing the step's tag (i.e. *i* and *t*). We call these *tag functions* since they are declared in terms of the step instance's tag. If the tile collection used "x, y" as the identifiers for its tag, the input tile would still be identified by "i, t" in the above declaration since the expressions are a function of the step's tag, not the item's key.

When the CnC graph (computation) is initialized, we need to provide all of the item instances for the initial time-step (*t*=0). Let us assume that we have a total of *NUM_TILES* tiles. We would update the initializer declaration as follows:

```
( $initialize: () )
 -> [ tile:      $range(0, NUM_TILES),   0 ],
    [ fromLeft:  $range(1, NUM_TILES),   0 ],
    [ fromRight: $range(0, NUM_TILES-1), 0 ];
```

Since the initializer is initializing the graph, it should only have outputs (no inputs). Note the built-in *$range* function's use in the output tag functions. We can use ranges to declare input or output relationships involving multiple step or item instances from the same collection. This is especially useful when the number of instances might be a dynamic count (in which case it wouldn't even be possible to enumerate each instance individually). The *$range(start, end)* syntax specifies the integer interval [start,

end). Note that since the first tile has no left input and the last tile has no right input, these are excluded from the ranges.

Our finalizer will collect all of the tiles from the final time-step, compute the average of the vector's values, and print the result. This will be a major bottleneck in our applications performance since all of the computation is being performed as a single task (this actually also applies to our initializer). It would be better to build a tree of computation tasks to do this reduction in parallel; however, for the sake of simplicity we will do all of the averaging in the finalizer. We update the declaration as follows:

```
( $finalize: () )
 <- [ tile: $range(0, NUM_TILES), LAST_TIMESTEP ];
```

We assume that LAST_TIMESTEP will be defined with the number of our last time-step. Again, we use the *$range* built-in to reference a dynamic range of item instances. Just as the initializer only declares outputs, the finalizer only declares inputs.

Control flow

We have our data, but the graph still does not describe when step instances are created. In CnC, we say that a step instance is *prescribed* when we "create" it, and that it is *ready* after it has both been prescribed and all of its inputs are available. We have already described the input dependences, but we still need to add the step prescriptions.

One option is to have the initializer prescribe all of the steps for all of the tiles and timesteps. We would declare that using two ranges:

```
( $initialize: () )
 -> ( stencil: $range(0, NUM_TILES), $range(1, LAST_TIMESTEP+1) ),
    /* item output declarations ... */ ;
```

This works fine for small computations. However, for a large computation, prescribing all of the steps at the beginning could overwhelm the runtime's task scheduler. We will stick with this method of prescription to keep our example simple. You can refer to the *4-improved.app* directory for an alternate graph specification that instead has stencil step instance prescribe the corresponding step in the next time-step.

Aliases for item instances

Notice that the stencil step has a tile, a left and a right item in both the input and the output. It might be helpful to add an alias to some of these instances to clarify, for example, which tile is the input and which is the output. This is obvious in the graph spec, but if we were to generate code using the current spec we would end up with identifiers like "tile0" and "tile1" for the two instances.

If we rename the output instance to "newTile" instead, then we will get the names "tile" and "newTile" in the generated OCR code. We will call the other outputs (used for the boundary exchange) "toLeft" and "toRight" We can give an item instance an alias by adding "*alias* @" in front of the item collection

name. (This syntax is borrowed from Scala and Haskell, which use @ to attach a name to a part of a pattern.)

The following is the full graph spec, including the conditionals as well as aliases for the output item instances in the stencil step declaration:

```
// 1-dimensional, 3-point stencil
// CnC-OCR tutorial example

// element tiles
[ float tile[TILE_SIZE]: i, t ];

// boundary elements (for exchange)
[ float *fromLeft:  i, t ];
[ float *fromRight: i, t ];

// graph initializer
( $initialize: () )
 -> [ tile:      $range(0, NUM_TILES),   0 ],
    [ fromLeft:  $range(1, NUM_TILES),   0 ],
    [ fromRight: $range(0, NUM_TILES-1), 0 ],
    ( stencil:   $range(0, NUM_TILES), $range(1, LAST_TIMESTEP+1) );

// stencil updater step
( stencil: i, t )
 <- [ tile:      i, t-1 ],
    [ fromLeft:  i, t-1 ],
    [ fromRight: i, t-1 ]
 -> [ newTile @ tile:      i,   t ],
    [ toRight @ fromLeft:  i+1, t ],
    [ toLeft  @ fromRight: i-1, t ];

// graph finalizer
( $finalize: () )
 <- [ tile: $range(0, NUM_TILES), LAST_TIMESTEP ];
```

Once you have saved the graph spec in Stencil1D.cnc then it is ready for the translator!

Code generation with the CnC-OCR graph translator

You can use the provided graph specification found in 1-base.graph as the starting point for this section.

The CnC-OCR graph translator utility (*cncocr_t*) parses the CnC graph specification, and then generates a full OCR project based on the given CnC graph structure. Run the following command to generate the project (in the current directory) from the graph spec:

```
cncocr_t Stencil1D.cnc
```

You should now see a bunch of files in our Stencil1D project. The project includes a makefile, and all of the code should build successfully right after generation if all the necessary definitions are provided. In our case we did not declare any special data types, but we do have three missing definitions: NUM_TILES, TILE_SIZE, and LAST_TIMESTEP. We can add these definitions to Stencil1D_defs.h (the only .h file in the current directory):

```
#define NUM_TILES 10
#define TILE_SIZE 10
#define LAST_TIMESTEP 100
```

After adding those definitions, the Stencil1D application should now build and run without any errors. You can build and run the application with this command:

```
make run
```

The translator tool should usually generate code that will compile and run to completion—although it will most likely give you the wrong answer since none of the computation has been implemented. However, our program hangs when we try to run it. This is a common symptom of a *missing dependence*. This occurs in OCR when an EDT dependence is never satisfied, and (equivalently) in CnC-OCR when one of a step's input items is never put. This type of problem can be difficult to debug since the details of the missing dependence are buried deep within the runtime internals.

Debugging tools

You can use the code provided in 2-base.app as a starting point for this section.

Luckily for us, the CnC-OCR toolchain includes some simple tools to help us diagnose why our program is hanging. First, we need to enable CnC debug event logging. Open the generated makefile, and uncomment the line that adds the CNC_DEBUG_LOG definition (the comment on the preceding line should say something like "Enable debug logging for x86"), then rebuild and run the application:

```
make clean run
```

The application will hang again, but this time you should see a file called *cnc_events.log* after you kill it. Analyzing the event log with the provided script should help us figure out what is wrong:

```
debug_cnc_events.py cnc_events.log
```

This will print a list of all the stalled steps, noting which inputs are missing (and causing the program to hang). It is usually a good strategy to look at the earlier timesteps first, since if those do not run, their missing outputs cause later steps to stall as well. There are two stalled steps in timestep t=1, and both are missing one input. These inputs correspond to the edge elements that are fixed with the value 1 in our stencil. We never create these items, which explains why the program hangs!

There are a few approaches we could take to fix this bug. We could put dummy items with the value 1 for all timesteps, which would satisfy the dependences. Another option is to write a separate step function to handle the edge cases. A third option is to mark the inputs as conditional, where the conditions disable the offending inputs on the first and last tile. We will use the last approach.

The graph syntax provides the built-in conditional *$when(cond)*, which can appear after a step or item instance reference to make it conditional. We can add conditionals to stop recursive prescriptions when *t=LAST_TIMESTEP*, and turn off the left or right values on the edge tiles. Adding conditions to disable inputs on the first and last vector elements (the fixed elements) results in the following:

```
 ( stencil: i, t )
 <- [ tile:      i, t-1 ],
    [ fromLeft:  i, t-1 ] $when(i > 0),
    [ fromRight: i, t-1 ] $when(i+1 < NUM_TILES)
 -> /* output item instances ... */ ;
```

If you run the translator tool again, and build, the program should now run to completion; however, since we have yet to implement the step code it still does not give any useful output.

Note: Remember to disable the debug logging now.

Fleshing out the skeleton code

The skeleton code is littered with TODO items. A quick search for TODO gives matches in the following files: Main.c, Stencil1D.c, Stencil1D_defs.h, and Stencil1D_stencil.c. The TODO items in Main.c and the header file just deal with the struct used to pass custom arguments into the graph. We have no custom arguments, so we can remove those TODO items. That means we just need to edit Stencil1D.c (which contains the initializer and finalizer) and Stencil1D_stencil.c (which contains the stencil step). We can start with the stencil computation in Stencil1D_stencil.c.

The first TODO item says to initialize *nextT*, which is the updated tile. We will need to loop over the tile (array) to do the 3-point stencil computation to produce each updated element value. Since we will have to handle the edge cases outside of the loop, we should define a macro to avoid repeating the stencil computation arithmetic. As per our original stencil equation:

```
#define STENCIL(left, center, right) \
        (0.5f*(center) + 0.25f*((left) + (right)))
```

In the case that either *left* or *right* was disabled by the corresponding input condition, the provided pointer value is NULL. Since we want to use the constant 1 in the case when there is no left neighbor, we can use the following line of code to compute the first value in our tile:

```
// first (conditional, default=1)
const float first = fromLeft ? *fromLeft : 1;
newTile[0] = STENCIL(first, tile[0], tile[1]);
```

The ternary conditional expression uses the *left* value if one was provided, or 1 otherwise. The inner elements are computed in a loop, and the final element is computed similarly to the first:

```
// inner
for (j=1; j<lastJ; j++) {
    newTile[j] = STENCIL(tile[j-1], tile[j], tile[j+1]);
}

// last (conditional, default=1)
const float last = fromRight ? *fromRight : 1;
newTile[lastJ] = STENCIL(tile[lastJ-1], tile[lastJ], last);
```

The remaining two TODO items tell us to initialize *nextL* and *nextR*, which are just the first and last elements in the array. Note that *nextL* goes to the right-hand neighbor (not the left). These two TODO items can be completed with these two initializations (on their respective lines):

```
*toRight = newTile[lastJ];

*toLeft = newTile[0];
```

The stencil step is now fully implemented. We can now continue with the initializer and finalizer functions defined in Stencil1D.c.

There are TODOs in the initializer for initializing *tile*, *left*, and *right*. The memory for all of these should just be set to zero. You can use a loop or *memset* for *tile* since it is an array. The *left* and *right* item values can just be set to 0 directly (e.g. **fromLeft = 0;*) since they are scalar values.

In the finalizer function, we want to compute the average of all the vector elements, and then print that average as our final result. Since the input to the finalizer was specified as a range of tiles, we end up with an array of tiles (*float **tile*) as the input. The graph translator tool generated a loop for accessing each tile in the array, and included a TODO item reminding us to do something with each tile.

```
void Stencil1D_finalize(float **tile, Stencil1DCtx *ctx) {

    int i, j;
    double total = 2; // first and last are always 1

    for (i = 0; i < NUM_TILES; i++) {
        for (j = 0; j < TILE_SIZE; j++) {
            total += tile[i][j];
        }
    }

    double count = 2 + TILE_SIZE * NUM_TILES;
    printf("avg = %f\n", total/count);

}
```

Getting an answer

You can use the code provided in 3-working.app as a starting point for this section.

Now the program is fully implemented! Running the program should now yield an average value:

```
$ make run
avg = 0.119571
```

Exercises

1. Play with different tile sizes, vector sizes, and numbers of timesteps.
2. Eliminate the unused items put by the computation.
3. Study the code in *4-improved.app* for a variant implementation:
    a. Using a global graph context to store graph parameters (e.g. tile size)
    b. Reading graph parameter values from the command line arguments
    c. Prescribe steps iteratively (rather than all prescribed by the initializer)
    d. Basic memory management