Report Week 6

Achievements:

Right now, 16 vertices on each chip form a ring structure with edges, which looks as follows:

```
 ---------------------------------
|                                 |
|    [02]--[03]--[04]--[05]--[06] |
|     |                       |   |
|    [17]                    [07] |
|     |                       |   |
|    [16]                    [08] |
|     |                       |   |
|    [15]                    [09] |
|     |                       |   |
|    [14]--[13]--[12]--[11]--[10] |
|                                 |
 ---------------------------------
```

After wiring up the edges, I implemented a simple function that counts the total number of data entries (the number of rows to be more precise) in all cores. Core 02 starts the entire process by sending a MCPL packet containing its number of data entries to core 03. Core 03 increments the received message by its number of data entries and sends it to core 04. The process repeats until core 02 is reached, which will write the content of the MCPL packet back to SDRAM:

```
2017-11-02 16:49:38 INFO: Time 0:00:00.369579 taken by RouterProvenanceGatherer
Getting profile data
|0%                          50%                         100%|
 ============================================================
2017-11-02 16:49:38 INFO: Time 0:00:00.008371 taken by ProfileDataGatherer
2017-11-02 16:49:39 INFO: 0, 0, 2 > 2
2017-11-02 16:49:39 INFO: 0, 0, 3 > 4
2017-11-02 16:49:39 INFO: 0, 0, 4 > 6
2017-11-02 16:49:39 INFO: 0, 0, 5 > 8
2017-11-02 16:49:39 INFO: 0, 0, 6 > 10
2017-11-02 16:49:39 INFO: 0, 0, 7 > 12
2017-11-02 16:49:39 INFO: 0, 0, 8 > 14
2017-11-02 16:49:39 INFO: 0, 0, 9 > 16
2017-11-02 16:49:39 INFO: 0, 0, 10 > 18
2017-11-02 16:49:39 INFO: 0, 0, 11 > 20
2017-11-02 16:49:39 INFO: 0, 0, 12 > 22
2017-11-02 16:49:39 INFO: 0, 0, 13 > 24
2017-11-02 16:49:39 INFO: 0, 0, 14 > 26
2017-11-02 16:49:39 INFO: 0, 0, 15 > 28
2017-11-02 16:49:39 INFO: 0, 0, 16 > 30
2017-11-02 16:49:39 INFO: 0, 0, 17 > 32
```

In this case, I put two data entries on every core before running the algorithm

Documentation (C code):

Disclaimer: This particular code is designed for the ring structure (specified above).
Different topologies require their own set of functions.

void start_processing()
//Given the header.function_id (supplied in the SDRAM), start_processing() selects
//which function to execute. Currently 1 → count, 2 → builds index table

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// COUNT FUNCTION                                                                                             //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

void count_function_start()
//if header.initiate_send is 1, this particular vertex is entitled to start the entire process
//by sending the first MCPL package to its neighbour in the ring
//The message is a 32bit integer that represents the number of rows in the data table
//on this particular vertex

void count_function_receive(uint payload)
//if header.initiate_send is 0, we have not reached our original vertex yet -
//therefore we increment the message by the number of rows in the data table on this
//particular vertex and send it to the next vertex. Then, we record the message to
//SDRAM, so that it can be seen in the output later on

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// BUILD INDEX TABLE FUNCTION                                                                                 //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The main idea of this function is to assign a unique identifier of size 4 (bytes) to each unique string data item – while making sure that all of those data items are uniquely identified across all 16 cores within the ring without inconsistencies, such as the same data item having different identifiers on different cores.
String data entries consist of 16 bytes and can only be sent in chunks of four – clogging up spike traffic and slowing everything down. By putting the unique identifiers into place, information regarding the entries can be shared much quicker, since now only the 4 byte identifiers have to be passed around.
Furthermore, the main vertex (or the leader) will keep a dictionary of all indices and data entries in its memory. This will enable checking whether certain data entries exist and what type of id they use within this specific ring network. Once we get to the stage where several rings will be linked together, each leader has to know exactly what information is stored within the network that it is assigned to, be it the current ring or some other form of topology in the future.

```
struct index_info {

      unsigned int *id_index;
      /* Holds the unique identifier for each data entry
       * Example: id_index[1] contains the unique id for the
       * second data entry within SDRAM
       * Currently this works only for one column
       * Length: header.num_rows
       */
      unsigned int *message;
      /* Holds 4 integers that make up a string
       * Designed to take a string entry that has been forwarded
       * by 4 distinct MCPL packages
       */
      unsigned int  message_id;
      /* Holds the unique id of string above
       * Takes the id from an incoming MCPL package as well
       */
      unsigned int  messages_received;
      /* Keeps track of number of MCPL packages received
       * if messages_received mod 5 = 0, a string data entry and its
       * id have been received
       */
    unsigned int  index_complete;
      /* A flag that tells if the index on this vertex is complete
       * Complete = 1; Incomplete = 0;
       * Complete means that there are no indices left with value 0
       */
      unsigned int  max_id;
      /* Tells you the highest id number on this vertex
       */

      unsigned int message_0000_sent;
      /* A flag that tells you if this vertex already
       * sent 0-0-0-0 to its neighbour
       * If that is the case, all the vertex has to do upon receiving
       * messages is to forward without invoking update_index()
       */
};

struct index_info local_index;
```

All index information is available globally through “struct index_info local_index”

void initialise_index()
//The first function that is executed by every core before any MCPL packages have
//been sent.

//Step 1: All vertices allocate memory to their index
→ local_index.id_index = malloc(sizeof(unsigned int) * header.num_rows)

//Step 2: All vertices create a buffer for receiving string messages
→ local_index.message = malloc(sizeof(unsigned int) * 4)

//Step 3: If this is the vertex leader, (meaning header.initiate_send == 1):
invoke function complete_index → builds an index table on this vertex

//Step 3: If this is a normal vertex:
Set all Ids to 0 in “local_index.id_index[i]” for all i

void complete_index(unique_id, start_index)
- param 1: unique_id → allows you to specify a specific id to start with
- param 2: start_index → where to start modification of id_index array

//Goes through all entries in SDRAM via
//“address = data_specification_get_data_address()”
//and put unique IDs into the “local_index.id_index[i]”
//Every ID in “local_index.id_index[i]” corresponds
//to an entry in the SDRAM in the given order.
//The algorithm makes sure that no two identical data entries get different Ids
//Also, the algorithm still works if only part of id_index has been completed,
//that is some entries still have the id 0 assigned to them
//Eventually, local_index.index_complete is set to 1

void update_index()
//Only executes if no [0-0-0-0,id] message has been sent by this vertex so far
//This function takes the newly arrived message from “local_index.message” and
//“local_index.message_id” and checks if any string data entries within its section of
//the SDRAM memory are identical with “local_index.message”. There are several
//scenarios:
//
//Scenario 1: There are data entries which are identical with local_index.message.
Their ids are still 0.
→ The ids of the data entries are updated to local_index.message_id
//
//Scenario 2: There are data entries identical with local_index.message.
Their ids are not 0 anymore.
→ The message has gone through all cores already.
Invoke message_reached_sender() to handle this

void index_receive(payload)
- param 1: payload → 32bit integer from incoming spike message

//Whenever a string data entry is being sent to the current vertex, the vertex waits
//until it receives all 4 MCPLs for the string entry and the one MCPL for the id.
// Once all information has arrived, it is stored in “local_index.message” and
//“local_index.message_id”.

//Scenario 1: This is a normal message
- → The function invokes “update_index()”
- → Once finished, the message is forwarded to the next vertex alongside its id

//Scenario 2: The message consists of four 0 integers and an id
- → This means that the previous vertex has finished synchronising its ids over the network
- → complete_index() is invoked if there are any ids left in the index that are 0, then the first item with a newly assigned id is sent to the next vertex
- → If there are no 0's within the id_index, that actually means that all ids have been assigned through update_index() in the past - therefore, no new ids can be assigned. The vertex forwards the [0-0-0-0,id] and sets message_0000_sent to 1
- → If message_0000_sent is 1 already upon reception, that means that all vertices have this flag set to 1 → the entire synchronisation is finished

void index_message_reached_sender()
//This function is invoked upon the arrival of a message that has originally been
//sent by this vertex. More precisely, this means that the message already visited all
//other vertices within the ring. Therefore, the message's id in the local vertex is
//not 0 anymore.

//Scenario 1: The message's id is smaller than the max_id of the local index
- → new_id = message's id + 1
- → use new_id to retrieve the corresponding string data item
- → send the string data item with new_id to the next vertex

//Scenario 2: The message's id is exactly the same a max_id
- → this means that all the ids in this vertex got assigned to their corresponding data items locally and elsewhere. In other words, all local ids have been synchronised within the ring/network
- → There still might be ids in other vertices that do not exist in this vertex which still need to be synchronised across the network
- → in order to indicate that this vertex has finished synchronising, a [0-0-0-0, max_id+1] message is sent to the next vertex

This is a visualisation of the data structures built by the functions above:

- The 'Data Entry' column is stored within SDRAM (acessed through the addresses)
- The 'Unique ID' column is stored in local_index.id_index on every vertex/core in the tightly coupled RAM, which can hold up to 32Kb of memory

Vertex [02] – Leader Index Table	
Data Entry	**Unique ID**
United Kingdom	1
Germany	2
France	3
Germany	2
United Kingdom	1

Vertex [02] – Leader Lookup Table	
Data Entry	**Unique ID**
United Kingdom	1
Germany	2
France	3
Spain	4
Greece	5

Vertex [03] – Chain Index Table	
Data Entry	**Unique ID**
Spain	4
Spain	4
Germany	2
Germany	2
Germany	2

Vertex [04] – Chain Index Table	
Data Entry	**Unique ID**
Greece	5
Spain	4
United Kingdom	1
Germany	2
Germany	2