MERCATOR: Mapping EnumeRATOR for CUDA [1]
User Manual

3/27/17 [2]

Stephen V. Cole (svcole@wustl.edu)
Jeremy D. Buhler (jbuhler@wustl.edu)

Washington University in St. Louis
St. Louis, MO, USA

[1]Funded by NSF CNS-1500173 and Exegy, Inc.
[2]The latest version of this manual is available at `http://sbs.wustl.edu/pubs/MercatorManual.pdf`

Contents

Introduction

Welcome to the MERCATOR user manual! MERCATOR is a CUDA/C++ system designed to assist you in writing efficient CUDA applications by automatically generating significant portions of the GPU-side application code. We hope you find it helpful; please feel free to contact the authors with any questions or feedback.

1.1 Intended use case: modular irregular streaming applications

MERCATOR facilitates the development of applications that are

- *streaming*: they take many small data items as inputs, process them once, and output or discard them;
- *modular*: they may be described by a (constrained) Data Flow Graph of processing steps (i.e., nodes) connected by directed data flow edges; and
- *irregular*: any node may filter an input item or replace it with new output item(s) to send downstream, and the data flow triggered by any given input is unknown at compile time.

MERCATOR's support for irregularity distinguishes it from systems that process Synchronous Data Flow (SDF) applications, in which the I/O rate(s) of each node are known *a priori*.

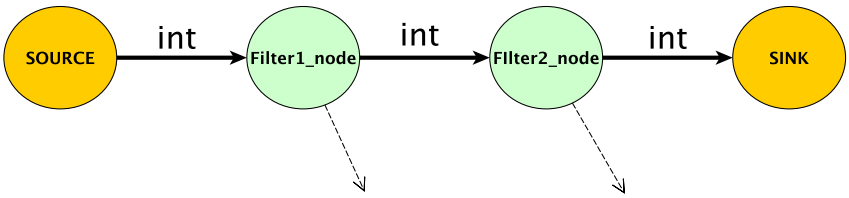


Figure 1.1: Example Data Flow Graph of an application with two filtering nodes that each accept integers and output integers. Dashed lines indicate that input items may be dropped by the filtering nodes, which makes the data flow of this application irregular. The `SOURCE` and `SINK` nodes shown represent input generation and output collection in the application.

Figure 1.1 shows the DFG of a simple filtering application.

1.2 MERCATOR application terminology

We use the following terminology to describe the topology of a MERCATOR application:

- A *module type* (or just *module*) is an entity whose type is defined by its function (code).
- A *module instance* is one occurrence of a module in an application's DFG (i.e. one *node*).
- An *edge* represents data flow between two module instances. Note that due to their support for irregular data flow, the exact data movement cardinalities of edges are not necessarily known at compile time.
- Data inputs flow into the application from a single `source node`, and outputs may be emitted to one or more `sink nodes`.

MERCATOR supports any modular streaming application whose topology obeys the following constraints:

1. Each instance may have multiple output edges but only a single input edge (not counting any *back edges*; see below).

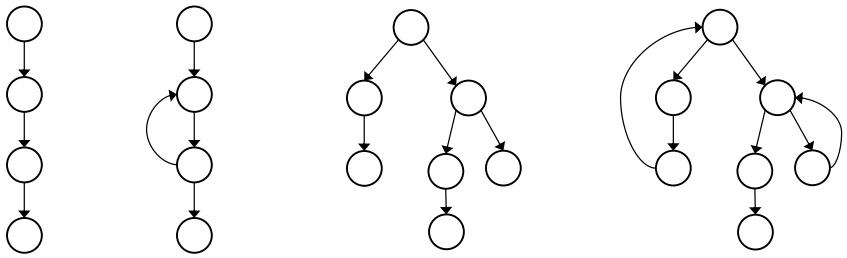


Figure 1.2: Examples of valid MERCATOR application topologies, including pipelines and trees with or without limited back edges.

2. While input items may trigger the generation of multiple output items from a node, a maximum number of possible output items generated for each input item is known at compile time.

3. Back edges may exist between a node and one of its ancestors, creating a loop. Self-loops are permitted; overlapping or nested loops are not. For each node in a loop, the output channel that participates in the loop must produce at most one output per input to the node.

Examples of valid application topologies are shown in Figure 1.2.

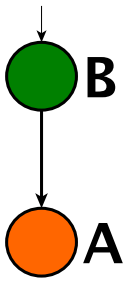
We use the following terminology to describe additional aspects of MERCATOR:

- Any code generated by MERCATOR is considered *system* code, while any code supplied by an application programmer (the *user* of the system) is considered *user* code.
- User-supplied *parameter data* may be associated with MERCATOR DFG objects at three granularities: a global set of data, module-type-specific data, and node-specific data. This allows nodes, module types, and the overall application to maintain state during execution.

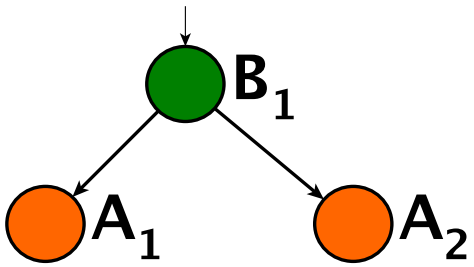
When implemented in CUDA, the code for each module type is a separate function, with module instances of the same type sharing function code but operating on different input sets with potentially different parameter values. This sharing of code between instances is important in the CUDA context where execution happens in wide SIMD. In fact, one key

contribution of MERCATOR's execution model is to group work by module type rather than by node to take maximal advantage of CUDA's wide-SIMD execution.

Example DFGs labeled with MERCATOR module types and instances are shown in Figure 1.3.



(a) DFG with two instances, each of a different module type indicated by its label and color.



(b) DFG with three instances of two module types. Subscripts in the type label distinguish instances of the same type.

Figure 1.3: Example DFGs. `SOURCE` and `SINK` nodes have been omitted for clarity. Note that when executing the graph in (b), MERCATOR may fire instances A_1 and A_2 simultaneously since they are of the same module type.

1.3 System overview and basic workflow

Given the terminology in the previous section, the application support provided by MERCATOR may be stated as follows: the MERCATOR *system* supplies CUDA code for a DFG's *edges* and provides an interface between the *edges* and *nodes*, enabling a *user* to generate a fully-functional CUDA implementation of a DFG by supplying code for its *module types* only. We refer to this implementation as a 'MERCATOR application' or just 'application,' even though it is not a full runnable CUDA program without host-side driver code that launches the device-side MERCATOR application. MERCATOR provides wrapper functions that perform these kernel calls and exposes the wrapper functions to the user via a CPU-side (host-side) API.

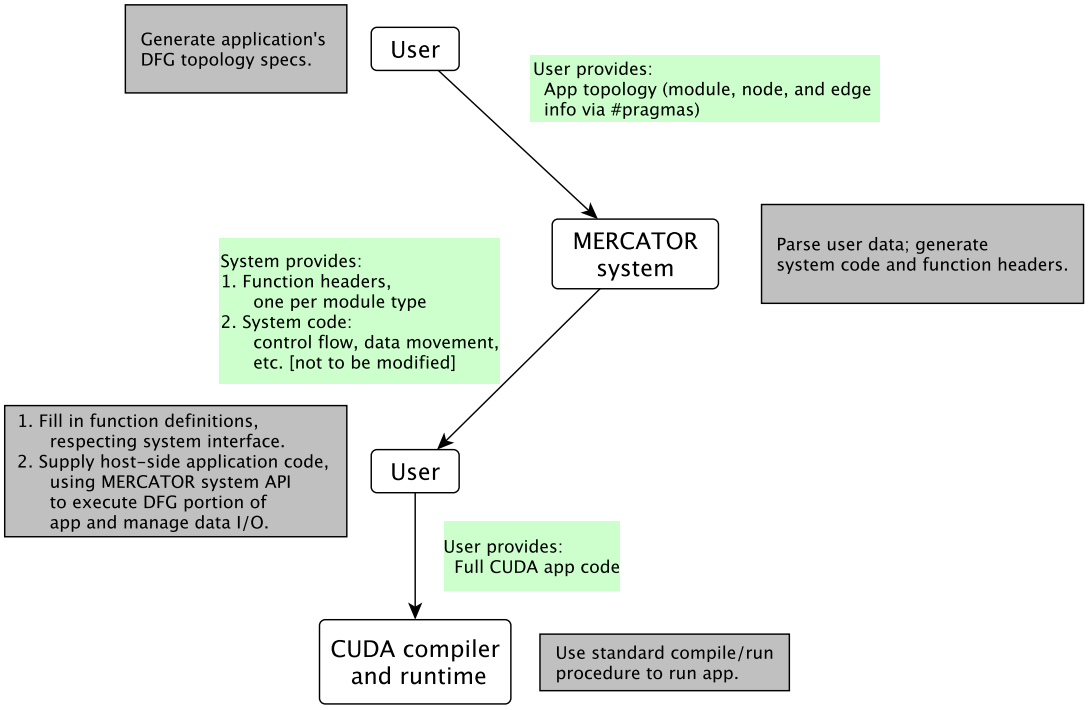


Figure 1.4: Workflow for writing a program incorporating a MERCATOR application.

The user interacts with a MERCATOR application via this host-side API, which in addition to the function calls that initialize and run the MERCATOR application contains functions for providing inputs and gathering outputs from it. A MERCATOR application is therefore deployed in a fully user-defined context.

In order to generate edge code, MERCATOR must have information about an application's topology, which the user provides to the system in a well-defined specification format. In order to maintain a consistent interface between nodes and edges, MERCATOR provides the user with function headers for each module type in the application.

An overview of the MERCATOR workflow is shown in Figure 1.4. Each phase of the workflow is described in the system documentation that follows.

MERCATOR system documentation

In the following instructions, we highlight the user's responsibilities and the interface between the user and the MERCATOR system.

2.1 Initial input

We begin with the user's initial input to the system, which consists of `#pragma` statements specifying the application's name, topology, module type requirements, and user-defined parameter data types.

2.1.1 Application name

The name of an application is provided by the user in a single *app specification* statement that must conform to the following EBNF grammar. Note that here and in all following specifications, we use the shorthand `<typename>` to stand in for the set of the names of all CUDA-valid types, including user-defined types.

```
   <module-spec>  =  #pragma mtr application <appname>
                     { "<" <parameter-type> ">" }
       <appname>  =  <string>
<parameter-type>  =  <typename>
```

Here `<appname>` is a string representing the name of the application and `<parameter-type>` is the type name of (optional) parameter data to be associated globally with this application.

2.1.2 Module type specification

Information about each module type in the application– which may be multiply instantiated in its Data Flow Graph– is provided by the user as a *module specification*. Specifications for user-defined module types must conform to the following EBNF grammar.

```
      <module-spec>  =  "#pragma mtr module" <module-name>
                         { "<" <parameter-type> ">" } "(" <input-stream> "->"
                        <output-stream>{"," <output-stream> } "|"
                        <elements> ":" <threads> ")"
     <input-stream>  =  <io-type> <num-inputs>
    <output-stream>  =  <stream-name> "<" <io-type> ">"
                        [":" <outputs-per-input>]
   <parameter-type>  =  <typename>
       <num-inputs>  =  "[" <int> "]"
<outputs-per-input>  =  <int> | ("?" [<int>])
      <module-name>  =  <string>
      <stream-name>  =  <string>
          <io-type>  =  <typename>
         <elements>  =  <int>
          <threads>  =  <int>
```

Explanation:

The specification consists of three main parts: a description of the data type and range of item cardinalities for each input and output stream, and the work assignment ratio. Notes on specific symbols follow.

`<parameter-type>` : type name of (optional) parameter data to be associated with this module type.

`<num-inputs>` : maximum number of input elements executed simultaneously upon each module firing. This does not limit the total number of input elements executed per module firing, only the number executed simultaneously in SIMD.

`<outputs-per-input>` : number of output elements produced per input element. A "`?`" indicates that the number of outputs per input is *a priori* unknown but bounded above by the integer specified in `<outputs-per-input>`. The absence of a "`?`" indicates a fixed known data rate. If no integer is provided, the number of outputs per input is assumed to be 1; for example, "`?`" is equivalent to "`?1`".

`<module-name>` : chosen by user.

`<stream-name>` : chosen by user. Required to disambiguate output streams.

`<elements>` : `<threads>` : ratio of input elements to threads for processing.

`<num-instances>` : total number of instances of this module type in application's dataflow graph

When more output elements are produced than inputs consumed per firing, outputs are produced on a per-element basis; that is, each input element leads to the production of (up to) a fixed uniform number of output elements. This uniform bound is the integer specified in `<outputs-per-input>`.

The current version of MERCATOR only supports an `<elements>:<threads>` ratio with `<elements>` = 1. In other words, multiple threads may be assigned the same input item during execution of a module, but one thread may not be assigned multiple input items.

Examples:

1. Vector addition module:

```
struct VectorPair {
 Vector a, b;
 };
#pragma mtr module addVec(VectorPair[128] ->
outStream<Vector> | 1:1)
```

   Interpretation: The module processes up to 128 input elements per firing, each input element being a struct containing two vectors and each output element being a single vector. The absence of a specified number of outputs per input indicates that each input produces a single output, no matter how many inputs are processed by a given firing. The module assigns one thread to each input element, which implies that the entire element-wise sum of the two vectors is computed by a single thread.

2. Vector addition module, different mapping ratio:

```
#pragma mtr module addVec(VectorPair[128] ->
          outStream<Vector> | 1:4)
```

   Interpretation: Here we assume the same definition of a `VectorPair` as above. This specification is identical to that of the previous example except that the module implementation assigns four threads to process each input vector pair. In this case, the user may arbitrarily divide the work of computing the vector sum among these four threads in the module function definition.

3. Filtering module:

```
#pragma mtr module evalThreshold<StateInfoStruct>(myItem_t[128] ->
          outStream<myItem_t> : ?  | 1:1)
```

   Interpretation: The module processes up to 128 input elements per firing, each element being of a user-defined type, and produces either 0 or 1 output elements per

input element. The module has associated with it a parameter data object of type `StateInfoStruct`.

4. Cloning module, single output stream:

```
#pragma mtr module cloneItem(myItem_t[128] ->
        outStream<myItem_t> : 4 | 1:1)
```

   Interpretation: The module processes up to 128 input elements per firing, each element being of a user-defined type, and produces 4 output elements per input element.

5. Cloning module, variable output production:

```
#pragma mtr module cloneItem(myItem_t[128] ->
        outStream<myItem_t> : ?4 | 1:1)
```

   Interpretation: Identical to previous example except that each input element produces between 0 and 4 output elements. In this case the MERCATOR infrastructure allocates space for the maximum number of output elements (i.e. 512) for each firing, and the user's module function definition is responsible for respecting this limit.

6. Module that combines cloning and filtering, multiple output streams:

```
        #pragma mtr module cloneFilterItem(myItem_t[128] ->
outStream1<myItem_t>, outStream2<myItem_t>, outStream3<myItem_t> :?,
                    outStream4<myItem_t> :2   | 1:1)
```

   Interpretation: The module's output behavior varies by stream: it sends a single element to `outStream1` and `outStream2`, sends 0 or 1 elements to `outStream3`, and sends 2 elements to `outStream4` per input element per firing.

Special types: `SOURCE` and `SINK`

In addition to the application module types designed by the user, MERCATOR provides two module types that must be included in an application: `SOURCE` and `SINK`. The single

`SOURCE` module type serves as the data entry point into the application, taking input from a user-facing input buffer (to be discussed later) and feeding it to the conceptual root node of the application's DFG. The `SOURCE` module type is therefore always instantiated exactly once, and its input type and (single) output type always match the input type of the DFG's conceptual root node.

The `SINK` module type serves as a data exit point from an application, taking final output from user-defined nodes and passing it to user-facing output buffers (to be discussed later). Since the application's DFG may contain multiple nodes that produce final output, and each output stream may be of a different type, the `SINK` module type is parameterized by this output type and describes a category of types rather than a monolithic type. Each `SINK` type may be multiply instantiated, and its input and output type(s) will all match the final output type it is designed to pass through.

Although the implementation of `SOURCE` and `SINK` module types is part of MERCATOR's system code, the user must include `SOURCE` and `SINK` modules in the input specifications so that the system knows their input and output stream information and so that they may be appropriately included in the application topology. Since their functionality is limited, however, the amount of information required from their specification is considerably less than from user-defined module types, and their specifications are therefore simpler.

`SOURCE` and `SINK` module types must conform to the following EBNF grammar:

```
<source-module-spec>  =  "#pragma mtr module " ( "SOURCE" | "SINK" )
                          "<" <io-type> ">"
           <io-type>  =  <typename>
```

Here `<io-type>` is the input or output type processed by the module. As stated above, there can be only one `SOURCE` module specified for an application, but there may be multiple `SINK` modules specified if an application produces outputs of multiple types.

Examples:

1. `SOURCE` module:

```
#pragma mtr module SOURCE<int>
```

Interpretation: The `SOURCE` module, and hence the conceptual root of the application's DFG, accepts inputs of type int.

2. `SINK` modules:

```
#pragma mtr module SINK<int>                    (2.1)
#pragma mtr module SINK<MyItem>                 (2.2)
```

Interpretation: Two types of outputs may be produced by the application: integers and the user-defined type MyItem.

2.1.3 Node specification

Information about each module instance (i.e., node) in the application is provided by the user as a *node specification*. Node specifications must conform to the following EBNF grammar:

```
       <node-spec>  =  "#pragma mtr node " <node-name>
                       { "<" <parameter-type> ">" } ":" <module-name>
       <node-name>  =  <string>
  <parameter-type>  =  <typename>
<module-type-name>  =  <module-name>
```

Explanation:

The specification consists of two parts: a user-chosen name for the module instance, and the name of the module type of which it is an instance.

`<node-name>` : user-chosen name for this node that will be used to identify it throughout the application.

`<parameter-type>` : type name of (optional) parameter data to be associated with this node.

`<module-type-name>` : name of the module type of which this node is an instance; must match the `<module-name>` of some previously-declared module specification.

Example:

In the following example, the relevant module specifications are given, followed by the node specifications describing their instantiations.

```
filterStage1(myItem_t -> fs1outStream<myItem_t> :  ?  | 1:1)
filterStage2(myItem_t[128] -> fs2outStream<myItem_t> :  ?  | 1:1)
node1<ThresholdStruct> :  filterStage1
node2<ThresholdStruct> :  filterStage1
node3<ThresholdStruct> :  filterStage2
```

Interpretation: `node1` and `node2` are instances of the module type `filterStage1`, and `node3` is an instance of the module type `filterStage2`. Each node has an associated parameter data object of type `ThresholdStruct`.

`SOURCE` and `SINK` nodes

The specification of `SOURCE` and `SINK` nodes must obey the following EBNF grammar:

```
<source-node-spec>  =  "#pragma mtr node " <node-name> ":SOURCE"
  <sink-node-spec>  =  "#pragma mtr node " <node-name> ":SINK<"<io-type>">"
       <node-name>  =  <string>
         <io-type>  =  <typename>
```

Explanation:

Module instances (nodes) of the `SOURCE` and `SINK` module types are specified in the same way as nodes of user-defined module types, with the keyword '`SOURCE`' being used as the (unique) `SOURCE` module type, and the (possibly not unique) `SINK` module type being specified similarly but with its output type included.

Example:

In the following example, the relevant module specifications are given, followed by the node specifications describing their instantiations.

```
#pragma mtr module SOURCE
#pragma mtr module SINK<int>
#pragma mtr module SINK<MyItem>
...
#pragma mtr node sourceNode :  SOURCE
#pragma mtr node resultNode1 :  SINK<MyItem>
#pragma mtr node resultNode2 :  SINK<MyItem>
#pragma mtr node resultNode3 :  SINK<MyItem>
#pragma mtr node resultNode4 :  SINK<int>
```

Interpretation: `sourceNode` is the single source node, the nodes `resultNode1-3` produce application results of type `MyItem`, and the node `resultNode4` produces application results of type `int`. The specifications of all user-defined module types and nodes has been omitted in this example.

2.1.4 Edge specification

An *edge specification* describes a dataflow relationship between two nodes; i.e., it represents an edge in the application's Data Flow Graph. Edge specifications must conform to the following EBNF grammar:

```
    <edge-spec>  =  <us-node> "::" <output-stream>
                    "->" <ds-node>
      <us-node>  =  <node-name>
      <ds-node>  =  <node-name>
<output-stream>  =  <stream-name>
```

Explanation:

`<us-node>` : upstream node in connection; must match the `<node-name>` in a previously-declared node specification.

`<ds-node>` : downstream node in connection; must match the `<node-name>` in a previously-declared node specification.

`<output-stream>` : stream of `<us-node>` to be connected to input of `<ds-node>`; must match a `<stream-name>` in the specification of the module type of which `<us-node>` is an instance.

As a result of our single-input assumption for modules and prohibition of nested back-edges, each module may appear as the `<ds-module>` in only a single edge specification, except in the case of a back edge, in which it appears as the `<ds-module>` exactly twice: once with its parent node and once with its back-edge origin.

Note that since nodes of `SOURCE` and `SINK` module types are named in the same way as those of user-defined module types, edge specifications containing them need not have a distinct form. However, a user is expected to include the following `SOURCE` and `SINK` edge specifications in every application:

1. A single edge specification that explicitly connects the `SOURCE` node to the conceptual DFG root node.

2. An edge specification that explicitly connects each user-defined node that produces final application output to its associated `SINK` node.

Examples

In the following examples, the relevant module and node specifications are given, followed by the edge specification(s) describing the dataflow between the nodes.

1. Connection between two filtering modules:

```
filterStage1(myItem_t -> fs1outStream<myItem_t> :  ?  | 1:1)
filterStage2(myItem_t[128] -> fs2outStream<myItem_t> :  ?  | 1:1)
node1 :  filterStage1
node2 :  filterStage2
node1::fs1outStream -> node2
```

   Interpretation: `node1` and `node2` are instances of module types `filterStage1` and `filterStage2` respectively. The elements output by `node1` on its `fs1outStream` stream are placed on the input worklist of `node2`. *Note:* It is assumed that `filterStage1` and `filterStage2` instantiate different filter code, rather than the same filter code parameterized differently (e.g., a simple threshold filter with different threshold values). An example of such a topology exists in the Viola-Jones filter cascade, in which different filters assay different features of a candidate face image.

2. Connection requiring named streams for disambiguation:

```
splitItems(myItem_t[128] -> siOutStream1<int>:2,siOutStream2<int>:3 | 1:1)
filterItems(int[128] -> fiOutStream<int>:?  | 1:1)
node1 :  splitItems
node2 :  filterItems
node1::siOutStream1 -> node2
```

Interpretation: The elements in the `siOutStream1` output stream of `node1` are placed on the input worklist of `node2`. *Note:* In this case naming module output streams enables disambiguation; both `siOutStream1` and `siOutStream2` have the same type, so inferring which stream to use to connect `node1` and `node2` would not be possible without a stream ID.

Example: `LinearPipe` application

The set of module type, node, and edge specifications for an example MERCATOR application are shown in the Appendix in Listing 1. This application, which we refer to as "`LinearPipe`," implements the linear pipeline of filter nodes shown in the examples in Section 1 of this manual, with the additional property that all nodes in the pipeline may produce final output. This allows the application to expose elements that it filters out as well as those that pass through the filters at each stage. A diagram of the application's topology is shown in Figure 2.1. This application will serve as a running example for other pieces of user-generated code in the sections that follow.

2.2 System feedback

Based on the user's initial input, the MERCATOR system will generate and return CUDA code of two types: system infrastructure code and module function headers.

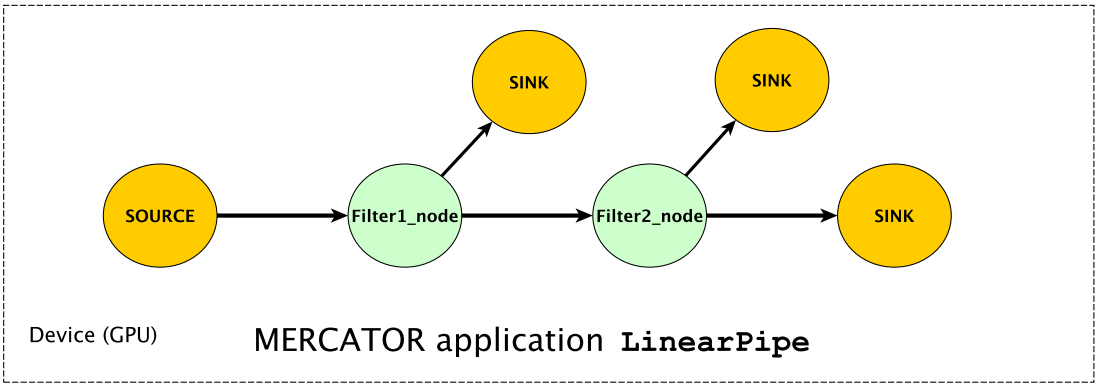


Figure 2.1: Diagram showing the DFG of the MERCATOR filtering application named `LinearPipe` described in the text. Note the existence of multiple `SINK` nodes.

The system infrastructure code is not intended to be modified by the user, but along with the user's function definitions, it constitutes an essential part of the MERCATOR application to be integrated into the user's final program code.

Module function headers

MERCATOR provides a set of module function headers written in CUDA that implicitly contain the user's interface to the MERCATOR infrastructure code. One header is produced for each module type, and the function bodies are to be filled in by the user. The details of writing function definitions are presented in the next section. Module function headers have the following format:

```
void run (
              <input-type> inputItem,
              int nodeIdx)
```

Explanation:

`<input-type>` : type of elements in module's input stream; determined by module specification previously given by the user.

`inputItem` : input element to be operated on by user code.

`nodeIdx` : an internally-generated tag for the current node being executed; used for API function calls in module definition.

Since MERCATOR's scheduling strategy involves firing multiple instances of the same module simultaneously, the `<instance-type>` as well as the `inputItem` may vary across threads executing the same function call.

2.3 Full program

2.3.1 Module run function definitions

If there are more threads present on the GPU device than there are items to be operated on, MERCATOR ensures that the "extra" threads do not execute the `run()` function call. This ensures that each thread that executes the function will have a valid input item passed in through the `inputItem` parameter; however, it also implies that to avoid undefined behavior, module definitions must be written with no CUDA synchronization calls such as `__syncthreads()`.

The remaining paragraphs in this section describe MERCATOR API functions accessible to the user inside `run()` functions.

Accessing user-defined parameter data

Within the `run()` function, users may access parameter data using the following API function calls:

- Application level: `get_appUserData()`
- Module type level: `this->get_userData()`
- Module instance level: `this->get_nodeUserData(nodeIdx)`

Here '`nodeIdx`' is the tag given to the user in the `run()` function header and is passed to the `get_nodeUserData` function verbatim.

Writing output

Output may be written from within the `run()` function using the following API call:

```
node->push(item, nodeIdx, Out::streamName)
```

where `item` is the output item to be written, `nodeIdx` is the tag parameter of the `run()` function, and `streamName` is the name given to the desired output stream by the user in the initial input specification. Here we reiterate that although many threads may be writing to the same output stream, no synchronization code should be included in the `run()` function; MERCATOR's system code ensures the integrity of the output streams.

Multiple threads per input element

When multiple threads are assigned to each input element (as dictated by the `<elements>:<threads>` ratio in the module type input specification), the following API call may be used to access a thread's offset relative to all threads assigned to the same input element:

```
eltOffset(threadIdx.x)
```

This call will return an `int`, which can be used to process the appropriate partition of a shared input. For example, if many threads are assigned the same input string of which they are intended to process consecutive substrings, each thread may find its starting point relative to the input string with a call to `eltOffset` .

Multiple output elements per input element

When multiple output elements may be written to an output stream for each input consumed, either by a single thread or in the case of multiple threads per input element, an output slot within the output stream must be specified for each `push` operation. This is done with an (optional) third parameter to the function, so that the function call will be of the following form:

```
node->push(item, 'nodeIdx', Out::streamName, slotIdx)
```

where `slotIdx` is the desired slot within the output stream relative to all possible outputs generated by the current input.

For example, if each thread writes four outputs to the same stream during each `run()` function execution, those four writes may take the following form:

```
node->push(item, 'nodeIdx', Out::streamName, 0) // the '0' parameter is optional
node->push(item, 'nodeIdx', Out::streamName, 1)
node->push(item, 'nodeIdx', Out::streamName, 2)
node->push(item, 'nodeIdx', Out::streamName, 3)
```

In the case of multiple threads being assigned to the same input element, the `eltOffset` function may be used to specify an output slot. For example, if multiple threads are assigned to the same input element and each thread writes a single output during each `run()` function execution, that write may take the following form:

```
node->push(item, 'nodeIdx', Out::streamName, eltOffset(threadIdx.x))
```

Multiple input elements per thread

When multiple elements are assigned to each thread (as dictated by the `<elements>:<threads>` ratio in the module type input specification), the module `run()` function header changes to

give the user access to all items assigned to the calling thread from inside the function. The revised header signature is:

```
void run (
              <input-type> *inputItems,
              unsigned char *nodeIdxs)
```

Items and tags are stored in arrays and may be accessed using array notation; e.g., `inputItem[0]` and `nodeIdxs[0]` hold the first item and tag assigned to the calling thread respectively.

Within the `run()` function, the following API call may be used to access the total number of elements assigned to a given thread:

```
this->itemsPerThread(threadIdx.x)
```

This call will return an `int`, which can be used to loop through the inputs and node tags passed in to the `run()` function. Note that the number of items assigned to a thread may not be the `numThreads` value specified in the module type input specification if the calling thread holds the (non-full) tail end of the set of inputs being executed by the module.

One possible set of function definitions for the headers produced for the `LinearPipe` application from the input specifications shown in Listing 1 is shown in Listing 2.

2.3.2 Application data I/O

The MERCATOR infrastructure code and the user's module definitions may be incorporated into a host-side driver file via a single `#include` of the module header file. The resulting CUDA code may be submitted to any CUDA compiler (e.g., `nvcc`) for compilation and subsequently executed using any CUDA runtime.

We now turn to the portions of the MERCATOR user interface that reside on the CPU. We begin with the data interface between user code and the MERCATOR application. MERCATOR provides special input and output buffer types to manage host-device data transfers, and it is these buffers with which user code interacts to move data into and out of MERCATOR. These buffer types are parameterized by the type of data they hold.

Input buffers

Input buffers are created with a capacity and, optionally, with an initial data set; the API signature for the input buffer constructor is the following:

```
InputBuffer<io-type>(int capacity, io-type* inputElements)
```

where `inputElements` is an optional parameter.

Note: The inputElements array MUST have been allocated using CUDA's "Managed Memory"– i.e., with a call to `cudaMallocManaged()` rather than the usual `cudaMalloc()`– so that the data will be accessible on both host and device.

If the initial data set is not provided, elements may be added to the buffer one at a time using the API member call `add(nextElement)`.

Output buffers

Output buffers are created with a capacity; the API for the output buffer constructor is the following:

```
OutputBuffer<io-type>(int capacity)
```

The application data stored in an output buffer is exposed the user as an array via the following member API functions:

```
io-type* OutputBuffer::get_data()
int OutputBuffer::size()
```

Data elements may also be retrieved from an output buffer one at a time with the API member call `get()` , which pops an item off the output buffer and returns it.

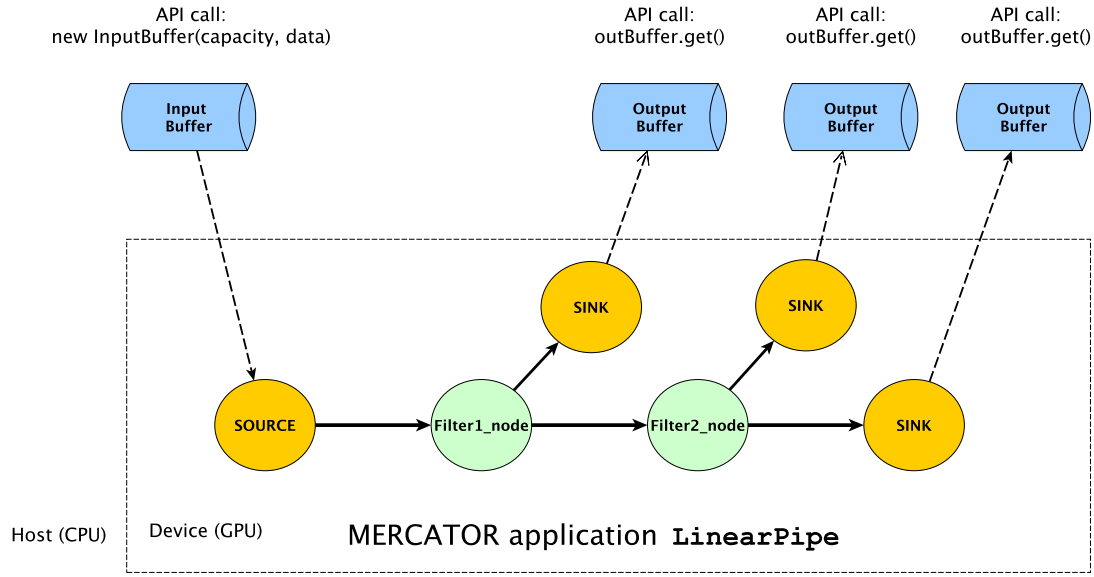


Figure 2.2: Diagram showing the `LinearPipe` DFG as well as its data interface with the host via input and output buffers.

A diagram of the `LinearPipe` application along with its associated input and output buffers is shown in Figure 2.2.

2.3.3 MERCATOR application hooks

To set up and run the MERCATOR application from the user's host-side program, the user creates an object of the application type given in the input specification, and API member calls on that object perform the necessary application-level operations.

Note: To ensure correct memory allocation using `cudaMallocManaged()`*, application objects must be created on the heap with a call to* `new()`*.*

Instantiated objects representing each module type and module instance of the application are provided and exposed to the user; these objects have the names given to each module type and instance in the user specification, and are typed as inner classes of the user-specified application class. These objects are invoked to associate I/O buffers with `SOURCE`/`SINK` nodes and to associate user-provided (non-stream) data with module types and instances.

User-provided data

The user may associate arbitrary data with a module, an instance, or the entire MERCATOR application, and this data may be accessed both from the user's host-side code and from within a module's `run()` function. Datatypes for these data must be given in the application specification file at the appropriate granularity.

Note: All user-provided data MUST be accessible in host and device memory. For non-POD data, this may be accomplished directly by the user by allocating all data using CUDA's "Managed Memory"– i.e., with a call to `cudaMallocManaged()` rather than the usual `cudaMalloc()`– or by defining the datatype to extend the `Mercator::CudaManaged` class and allocating the data on the heap with a call to `new()`.

User-defined data objects are bound to specific application objects with the following API calls:

- Application level: `appObject->set_userData(myAppData)`
- Module level: `appObject->ModuleTypeName->set_userData(moduleTypeNameData)`
- Node level: `appObject->set_nodeUserData(AppName::Node::NodeName, nodeNameData)`

Here `appObject` is the name of the application object created by the user; `AppName`, `ModuleTypeName`, and `NodeName` must match the names of these components given in the user's input specifications; and `myAppData`, `moduleTypeNameData`, and `nodeNameData` are user-defined data objects. `AppName::Node::NodeName` is an enum value used by MERCATOR to index the nodes within an application.

Inside each module's `run()` functions, the API calls to access user-defined data are given above in Section 2.3.1

Buffer/node association

In order to connect the buffers to the appropriate data entry/exit points of the MERCATOR application– i.e., the appropriate `SOURCE` and `SINK` nodes– the user must specify which buffer

gets associated with which node. This is done with an API call that operates on a node and takes the relevant buffer object as a parameter. The formats of the API member calls to set input and output buffers are the following:

```
sourceNodeObj->set_inBuffer(inBufferObj)
sinkNodeObj->set_outBuffer(outBufferObj)
```

where `sourceNodeObj`/`sinkNodeObj` are the user-supplied names of the source node and a sink node, respectively, and `inBufferObj`/`outBufferObj` are user-created input and output buffer objects. Because an application may include multiple output buffers, multiple calls to `set_outBuffer()` may be made for the same application, but only one call to `set_inBuffer()` should be made since an application includes only a single input buffer.

Running the MERCATOR application

Once input and output buffers have been created, the input buffer has been initialized with data, and buffers have been associated with nodes, the MERCATOR application may be run from a user program with the API member call `run()`. This blocking function call encapsulates the entire execution of the MERCATOR application, so that when it returns, the input buffer will be empty and the output buffers will contain all program outputs assuming an error-free execution on the GPU device.

Once the application has been run, data may be gathered from the output buffers using the API calls described above.

The relevant parts of the user program for the `LinearPipe` application are shown in Listing 3, and a diagram showing the full program layout is shown in Figure 2.3.

2.3.4 Restrictions

The following restrictions on incorporating MERCATOR applications into user programs currently exist:

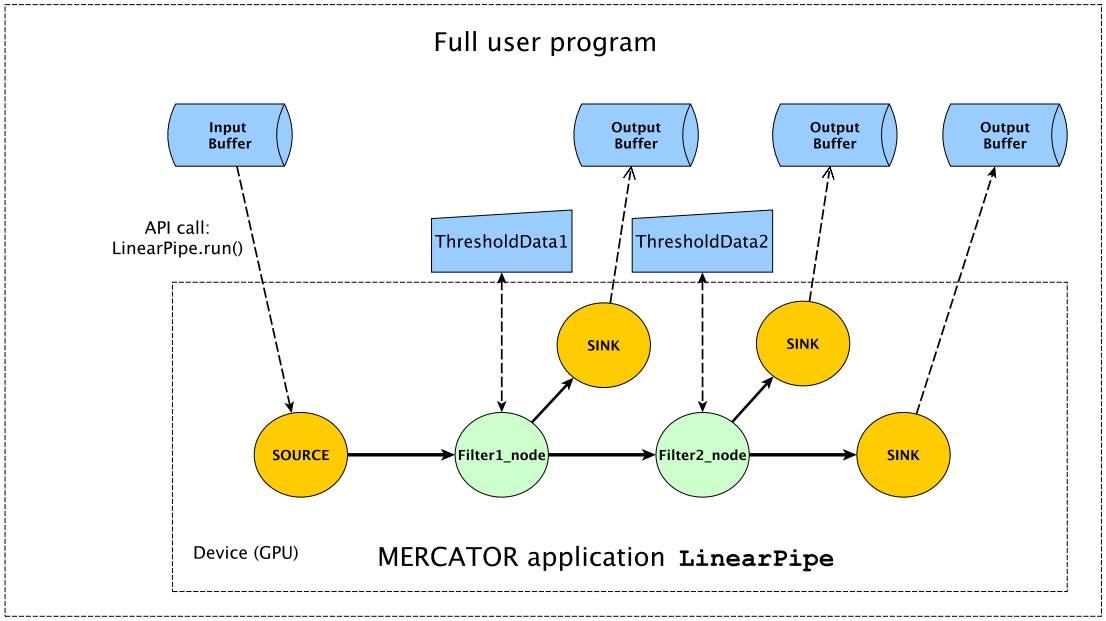


Figure 2.3: Diagram showing the `LinearPipe` DFG and its interface with host-side code and data, including I/O buffers and user-defined parameter data accessible from within the modules' `run()` functions. Two of the program's output buffers have been omitted for clarity.

- Single-batch execution: All input and output buffers must fit into the GPU card's global memory simultaneously; multiple rounds of execution and I/O copying are not supported.
- Single-stream execution: While multiple MERCATOR applications may be invoked in a single user program, each application must be invoked separately. This is automatically enforced by the internal semantics of the MERCATOR application `run()` function API call, which executes synchronously.

A Appendix: example application code

We show here all code required to create a simple filtering application in MERCATOR and access it from a user program.

A.1 Initial input specifications

Listing 1 shows the set of input specifications for the application.

Listing 1: Input specification for the "LinearPipe" MERCATOR application. The application contains 2 module types in addition to the `SOURCE` type and a single `SINK` type, and consists of a linear pipeline with shunts to `SINK` nodes off of each node in the pipeline.

```
/*** Application name (with app-level datatype name). ***/
#pragma mtr application LinearPipe<FilterAppData>

/*** Module (i.e., module type) specs. ***/

// Filter 1
// Note the user-defined parameter data of type "Filter1State"
#pragma mtr module Filter1<Filter1State> (
                int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 1 )

// Filter 2
#pragma mtr module Filter2 (
        int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 4 )

// SOURCE Module
#pragma mtr module SOURCE<int>

// SINK Module
#pragma mtr module SINK<int>


/*** Node (i.e., module instance) specs. ***/
#pragma mtr node sourceNode : SOURCE
#pragma mtr node Filter1_node : Filter1

// Note the user-defined parameter data of type "Filter2ThresholdStruct"
#pragma mtr node Filter2_node<Filter2ThresholdStruct> : Filter2
```

```
#pragma mtr node sinkNodeReject1 : SINK<int>
#pragma mtr node sinkNodeReject2 : SINK<int>
#pragma mtr node sinkNodeAccept : SINK<int>


/*** Edge specs. ***/

// SOURCE -> Filter1
#pragma mtr edge sourceNode::outStream -> Filter1_node

// Filter1 -> Filter2
#pragma mtr edge Filter1_node::accept -> Filter2_node

// Filter1 -> Rejects
#pragma mtr edge Filter1_node::reject -> sinkNodeReject1

// Filter2 -> SINK
#pragma mtr edge Filter2_node::accept -> sinkNodeAccept

// Filter2 -> Rejects
#pragma mtr edge Filter2_node::reject -> sinkNodeReject2
```

A.2 Module type definitions

Example module type definitions are shown in Listing 2.

Listing 2: Two sample module function bodies for the module types defined in Listing 1. `Filter1` conditionally writes four values either to the "`accept`" output stream or the "`reject`" output stream; note its use of `Filter1`'s parameter data, the global application parameter data, and the use of slot indices in the last parameter of the push() function. `Filter2` conditionally writes a single element to the `accept` or `reject` stream. Note its use of the `eltOffset()` function to assign output to the correct slot, since more than one thread is given the same input.

```
////  module Filter1<UserDataExt>(int[10] -> accept<int> : ?4, reject<int>: ?
__device__
void LinearPipe::DerivedModule_Filter1::run(int inputItem, unsigned char node
{
        int stateNum = this->get_userData()->stateNum;
        if(stateNum < 6)  {
          int scaler = get_appUserData()->scale;
          int outputItem = inputItem * scaler;
          node->push(outputItem, nodeIdx, Out::accept, 0);  // 0-param is opt
          node->push(outputItem + 1, nodeIdx, Out::accept, 1);
          node->push(outputItem + 2, nodeIdx, Out::accept, 2);
          node->push(outputItem + 3, nodeIdx, Out::accept, 3);
        }
        else  {
          node->push(inputItem, nodeIdx,     Out::reject, 0);
          node->push(inputItem + 1, nodeIdx, Out::reject, 1);
          node->push(inputItem + 2, nodeIdx, Out::reject, 2);
          node->push(inputItem + 3, nodeIdx, Out::reject, 3);
        }
    }

////  module Filter2(int[10] -> accept<int>: ?4, reject<int>: ?4 | 1 : 4)
__device__
void LinearPipe::DerivedModule_Filter2::run(int inputItem, unsigned char node
{
   if(inputItem < node->get_userData()->thresh)
     node->push(inputItem, nodeIdx, Out::accept, eltOffset(threadIdx.x));
   else
     node->push(inputItem, nodeIdx, Out::reject, eltOffset(threadIdx.x));
}
```

A.3 Driver code

Driver code for a user program that invokes the `LinearPipe` application is shown in Listing 3.

Listing 3: User code that creates MERCATOR I/O buffers, fills the input buffers with consecutive integers, initializes and invokes the "LinearPipe" application, and reads the results from the output buffers.

```
/**
* User program that acts as test harness
*      for MERCATOR filtering application "LinearPipe."
*
*/

#include <iostream>

#include "userInput/LinearPipe.cuh"  // headers for module run() fcns
#include "system_headers.cuh"      // other MERCATOR system headers

int main()
{
  ////// set up input buffer
  const int IN_BUFFER_CAPACITY = 128;

  int* inBufferData;
  cudaMallocManaged(&inBufferData,
    IN_BUFFER_CAPACITY * sizeof(int));
  Mercator::InputBuffer<int>* inBuffer =
    new Mercator::InputBuffer<int>(inBufferData,
        IN_BUFFER_CAPACITY);

  ////// set up output buffers
  // output buffers accommodate 4 outputs/input
  const int OUT_BUFFER_CAPACITY1 = 4 * IN_BUFFER_CAPACITY;
  const int OUT_BUFFER_CAPACITY2 = 4 * OUT_BUFFER_CAPACITY1;
```

```
Mercator::OutputBuffer<int>* outBuffer1 =
  new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY1);
Mercator::OutputBuffer<int>* outBuffer2 =
  new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY2);
Mercator::OutputBuffer<int>* outBuffer3 =
  new Mercator::OutputBuffer<int>(OUT_BUFFER_CAPACITY2);

////// fill input buffer
for(int i=0; i < IN_BUFFER_CAPACITY; ++i)
  inBuffer->add(i);

////// create app object
// NB: must be created on heap
LinearPipe* linearPipe = new LinearPipe();

////// set node-, module-, app-level user data
FilterAppData* fullAppData = new FilterAppData();
linearPipe->set_userData(fullAppData);

Filter1State* filter1data = new Filter1State();
linearPipe->Filter1.set_userData(filter1data);

Filter2ThresholdStruct* filter2nodeData =
  new Filter2ThresholdStruct(4);
linearPipe->set_nodeUserData(LinearPipe::Filter2::Filter2_node, filt

////// associate buffers with nodes
linearPipe->sourceNode.set_inBuffer(inBuffer);
linearPipe->sinkNodeReject1.set_outBuffer(outBuffer1);
linearPipe->sinkNodeReject2.set_outBuffer(outBuffer2);
linearPipe->sinkNodeAccept.set_outBuffer(outBuffer3);

////// run main function
```

```
    linearPipe->run();

    /////// retrieve output
  int* outData1 = outBuffer1->get_data();
  int* outData2 = outBuffer2->get_data();
  int* outData3 = outBuffer3->get_data();

  int outData1Size = outBuffer1->size();
  int outData2Size = outBuffer2->size();
  int outData3Size = outBuffer3->size();

    ////// print contents of output buffers
    for(int i=0; i < outData1Size; ++i)
      printf("Buffer 1 item %d: %d\n", i, outData1[i]);
    for(int i=0; i < outData2Size; ++i)
          printf("Buffer 2 item %d: %d\n", i, outData2[i]);
    for(int i=0; i < outData1Size; ++i)
      printf("Buffer 3 item %d: %d\n", i, outData3[i]);

  /////// clean up
  cudaFree(inBufferData);

  return 0;
}
```