1.0 StreamC Language Specification

The StreamC language, in combination with C++, is to be used for writing programs that utilize the Imagine stream processing system. StreamC includes commands for transferring streams of data to and from the Imagine system and between Imagine processors, for reading and writing microcontroller variables, and for executing kernels (written using KernelC). This document defines the StreamC language. It assumes basic knowledge of KernelC in general, and Imagine data types in particular.

1.1 Types

The primary purpose of StreamC is manipulation of stream data using stream types. StreamC also supports declaration of microcontroller variables (as per KernelC) which can then be read and written using special functions and passed as parameters to kernels.

1.1.1 Stream Types

Stream variables are analogous to pointers that refer to reference-counted data. Stream variables refer to a sequence of data elements indexed from zero. Stream variables are declared as follows:

stream <*type*> *name*;

where type is an imagine basic type or user defined record.

1.1.2 Imagine Basic Types in StreamC

Imagine basic types should be used in StreamC only to declare variables of stream types and microcontroller (UC) qualified types. To avoid name conflicts with normal C++ basic types, Imagine basic types are distinguished by an “im_” prefix in StreamC. For instance, the Imagine “int” basic type is “im_int”.

1.1.3 Imagine Basic Types in Shared Header Files

Kernels, records, and some StreamC functions are often declared in a headers file shared by both KernelC and StreamC. If such declarations are enclosed between “#include kernelc_deftypes.hpp” and “#include kernelc_undeftypes.hpp” directives then Imagine basic types are assumed and the “im_” prefix is not required. Otherwise, the “im_” prefix must be used for all Imagine basic types.

1.2 File Structure

StreamC is used in combination with C++ in the normal C++ file structure. StreamC requires the inclusion of the “streamc.hpp” header file. However, the IStream preprocessor, which optimizes StreamC, becomes less effective as the C++ surrounding the StreamC becomes more complex. In general, StreamC should be used in simple functions and stream variables should be declared directly (not dynamically allocated).

1.3 Streams

Stream variables are analogous to pointers to reference-counted data. When no stream variables refer to data, the space allocated to hold the data is freed. In the following text, “x”, “y”, and “z” are stream variables

1.3.1 Stream variable assignments

A stream variable can be assigned to refer to new data as follows:

x = newStreamData<*type*> (*[optional] size*);

If size is specified, a fixed number of stream data elements are allocated. If size is omitted, no elements are allocated, but any time a new element is written to the stream, size is increased to the include that element.

A stream variable can be assigned to refer to data referred to by another stream variable using standard assignment syntax:

y = x;

A stream variable can be assigned to refer to a subset of the data referred to by another stream variable. The simplest subset is a range. For example:

y = x(*start*, *end*);

assigns y to refer to all elements of x from *start* up to but not including *end*. For instance:

y = x(0, 8);

assigns y to refer to elements 0 through 7 of x.

More complex subsets can be achieved using one of three access types:

Strided access can be used to refer to every *stride*th element in the range, as follows:

y = x(*start*, *end*, access_stride, *stride*);

For instance:

y = x(0, 8, access_stride, 2);

assigns y to refer to elements 0, 2, 4, and 6 of x.

Bit-reversed access refers to every element in the range, but with the bits in the index rearranged using four steps:

1. divide by modulus
2. reverse order of bits in quotient
3. multiply bit-reversed quotient by modulus
4. add remainder of original division

Bit-reversed access is specified as follows:

y = x(*start*, *end*, access_bitreverse, *modulus*);

For instance:

y = x(0, 7, access_bit_reverse, 2);

assigns y to refer to elements 0, 1, 4, 5, 2, 3, 6, and 7 of x (in that order).

Indexed access refers to elements constrained within the range with indices given by the elements of a stream of unsigned integers, as follows:

y = x(*start*, *end*, access_index, *index*);

For instance, if the stream<im_uint> z, refers to elements with values 5, 6, 5, 4, 2 then:

y = x(0, 7, access_index, z);

assigns y to refer to elements 5, 6, 5, 4, and 2 of x (in that order).

All such assignments also accept two additional, optional parameters: *coordinate type*, and *record size.*

y = x(..., *coodinate type*, *record size*);

Coordinate type can be either “coord_records” (the default), indicating that the range start, end, stride, modulus, and index values are specified in records, or “coord_words” indicating that those parameters specified in words.

Record size (specified in words) can be used to override the default record size used when distributing the data to the clusters. Specifying a record size of 3 for a stream of integers, would result in cluster 0 receiving, elements 0, 1, 2, then 24, 25, 26, while cluster 1 received elements 3, 4, 5, then 27, 28, 29 and so on.

1.3.2 Stream length

The length of a stream is equal to the number of elements it refers to, and can be obtained as follows:

x.getLength();

1.3.3 Null streams

A stream is initially null -- it refers to nothing. A stream becomes valid when is is assigned to refer to some data. A stream may become null again when the special value NULL or another null stream is assigned to it.* The validity of a stream can be determined as follows:

x.isValid();

* Currently, a stream also becomes null when the stream it is derived from becomes null, but I think that is going to change.

1.4 Microcontroller variables

Microcontoller variables may be declared in StreamC as follows:

uc<*type*> name;

where type is an Imagine basic type (with the “im_” prefix).

Microcontoller variables may be read and written using the readUC and writeUC functions, as follows:

```
int i;
uc<im_int> uc_i;
uc_i = ucWrite(i);          // assign value of i to uc_i
i = ucRead(uc_i);           // assign value of uc_i to i
```

1.5 Kernels

A Kernel is essentially a function that operates on streams. See the KernelC for information on writing kernels. Kernels are called in StreamC just like any other function.

1.6 Special Functions

StreamC includes several special functions:

void **streamCopy**(im_istream<T> in1, im_ostream<T> out1);

copies elements from stream “in1” to stream “out1”.

void **streamLoadVect**(vector<Tv>& vect, im_ostream<T> out1);

loads elements from vector to stream “out1”.

void **streamSaveVect**(vector<Tv>& vect, im_istream<T> in1);

saves elements from stream “in1” to vector.

void **streamLoadFile**(char *file, String type, String args, im_ostream<T> out1);

loads elements from file to stream “out1”.

void **streamSaveFile**(char *file, String type, String args, im_istream<T> in1);

saves elements stream “in1” to file.

bool **streamCompareFile**(char *file, im_istream<T> in1, float threshold);

compares elements of stream “in1” to file with given threshold.

2.0 Sample StreamC

For now, the sample is fft, in the im_apps_template directory.

To port an application to StreamC do the following:

1. Copy all the _template files to your application directory, which must be in im_apps.

2. Rename all the copied _template files, replacing “template” with your application name.

3. Open the project file. Delete all of the _template files and replace with your renamed files. Search for “TODO” and follow the instructions.

4. Choose “Debug” configuration for degugging, “Imagine” to build.