hStreams Hands-On Tutorial

CJ Newburn and Jesmin Jahan Tithi, Intel

Welcome to hStreams!

hStreams was created to make it easier for you to expose and harvest task parallelism on heterogeneous platforms. This tutorial will walk you through code examples, build confidence, and help you toward mastery of the capabilities listed below.

In this document, we explain the conventions used. We've favored consistency and modularity over simplicity in the makefiles, so please don't get distracted by thinking that builds are complex; they're pretty simple.

Then we describe how to get set up (really important!), get ready and get going.

We hope this is helpful and fun.

I. Capabilities

A. Remote invocation

1. hello_hStreams_world

a) Header files

b) Initialization and finalization

c) Type-converting and packing parameters at the source

d) Invoking a sink-side function with a scalar variable

e) Creating a sink-side function

f) Building and running an hStreams application

2. pass_scalar_args_get_result

a) Add an additional scalar argument

b) Pass a pointer to the return value, and its size in the remote invocation

c) Enable the sink code to pass back a return argument

d) Add synchronization: Wait for the return value

3. pass_pointer

a) User allocation of memory

b) Wrap user memory in buffers, to convert heap addresses and maintain dependences, and to create a buffer instance at the sink

c) Transfer data from host to

d) Pass heap arguments in addition to scalar arguments

4. return_multiple_args

a) Use non-standard name and location of sink-side file

b) Return multiple arguments, in the source file

c) Return multiple arguments, in the sink file

5. simple_make

a) Learn the simplest way to compile and hstreams program shown in 1.hello_hStreams_world

b) Learn how to run the executable while properly setting up the lib paths, in the simplest way.

B. Basic use of streams, domains and buffers

1. straight_line_code_host

a) A straight line code, without tasks, as a starting point

b) Learn how to time sections of interest

2. task_code_host

a) Refactor the straight line code as tasks

b) Variables become arguments

3. task_code_sink_buffer_sync

Execute the tasks in hStreams

a) *Create 2 streams instead of 1*

b) *Wrap user-allocated memory in buffers*

c) *Transfer data in 2 streams*

d) *Enqueue a task*

e) *Insert cross-stream synchronization*

f) *Intra-stream dependences are resolved without explicit synchronization*

4. cross_buffer_xfer

a) *Transfer from one buffer to another*

b) *Eliminate a redundant copy operation*

c) *Take a different approach to synchronization*

C. Performance through tiling

0. compute_math_not_tiled_host
    a) *A compute intensive kernel that performs O(dimX*dimY*dimZ*length^3) computations on host.*
1. compute_math_tiled_host
    a) *A possible efficient way of tiling the original code*
    b) *Demonstrate indexing changes*
2. compute_math_not_tiled_hstreams_naive
    a) *Use hstreams*
    b) *Enqueue the entire compute function in one invocation in a stream*
    c) *Performs necessary data transfers to get the expected result.*
3. compute_math_not_tiled_hstreams_streaming
    a) *Split the original 3 nested outer loops onto two parts to use multiple streams. It keeps the outer most parallel loop on host and converts the body of the loop as a compute function.*
    b) *Enqueue this new compute function as tasks in the available streams in a round-robin fashion.*
4. compute_math_tiled_hstreams_streaming
    a) *Determine unique source proxy addresses that each outer loop iteration accesses. It also shows how intermediate addresses from a pre-allocated large buffer can be used as buffer addresses as well*
    b) *Initialize intermediate data at sink, only the portion being worked on*
    c) *Transfer output data from sink to source*
    d) *Tile the sink-side code following similar technique used in exercise #1, with address calculation adjusted based on the given intermediate address.*
5. compute_math_tiled_hstreams_multicard

In addition to all mentioned in 4, in this exercise you learn how to use hstreams to check the number of available MIC cards in the system and use all of them while doing the same work:

*a) Find out the number of physical domains and number of active physical domains using hStreams_GetNumPhysDomains and creates (num_phys_domains * streams_per_domain) streams using hStreams_app_init*

b) Chose the stream id in a round-robin using total collective number of streams in the modulus.

6. compute_math_not_tiled_hstreams_host_multicard

a) Include core API headers, needed for host-side domains and streams (for now)

b) First does a default initialization.

c) Get detailed info about each available physical domain.

d) Add a new logical domain in the system using the domain id, and CPU mask, based on the set bits on use_mask.

e) Create a new stream using a given stream id, the logical domain id where it should be created and a CPU mask.

f) Load-balance by choosing proper distribution of the threads

g) Set the hstreams with the newly set up options to enable using host by calling hStreams_SetOptions.

h) Call the compute function at the sink-side similarly irrespective of whether the sink side is a host

i) Add a host version of memset, which is still not available in hstreams (check the sink-side code)

j) See the Makefile: Use _host extension for host as target

NOTE: A good exercise for the user would be tile the sink side code. Furthermore, try to change number of domains and input size and check the performance difference.

II. Conventions

- We will work with two versions for each exercise
  - User code
    - start with this, build it toward the solution
    - has a _src.cpp suffix
    - has a backup in _src_original.cpp; you can start over with this if you wish, or ignore it
  - Solution code
    - has a _src_solution.cpp suffix
- Directory structure
  - Outline form: <letter>/<number> implies order
    - Each major idea has its own letter
    - Sequences that build up richness progress with numbers
    - File completion is your friend – you don't have to know the full name
  - Where user changes are required, solution directories are offered as a reference
- File format
  - Description of the point of each file
  - Each major section is delimited

    - Key points made for that file are marked with comments of the form //!!<letter>, where the letters correspond to the innermost bullet points in the list above.
- Building and running
    - Building will create
        - bin/host/NAME: executable for user and solution
        - bin/x100/NAME_mic.so and NAME_SOLN_mic.so: sink-side dynamic library
    - Each directory has the following
        - Makefile (identical)
            - Reads name.mk
            - Builds sink-side library, unless IS_MIC is not defined
        - names.mk (usually identical)
            - reads rootname.mk to create NAME
            - sets IS_MIC=1 unless there's no MIC version
            - derives NAME_SOLN, SINK_NAME, SINK_SOLN_NAME unless overridden
        - rootname.mk (unique) – contains only base name of directory
        - run.sh to run user code
        - solution_run.sh to run solution code

III. Getting set up

The tutorial code is in a tar file, tutorial.tgz. Install it to your local machine.

```
cd <AVAILABLE AREA, like /home/<your username> >
mkdir YOURDIR
cd YOURDIR
cp path_to_tutorial_tar/tutorial.tgz .
tar xf tutorial.tgz
```

You now have a *copy* of the tutorial and ref_code.

You need to get onto a node with the MICs. If you set up your environment before switching machines to get into a machine with MIC, you'll lose your changes and need to redo the same. In the new machine, since you have a new shell, so you have to get to your directory again.

```
cd YOURDIR/tutorial
```

Now set up your environment.

Firstly, if your shell is not a bash shell, change it to a bash shell by typing:

```
bash
```

Next do this from the tutorial directory. Ideally you should do this from the same place where the source_setup file is:

```
. source_setup <-- RIGHT! // If you are on a bash shell
```

This sets TUT_INSTALL to point to this directory – *this is critical for scripts to work.*

It loads the compiler module and sources the compiler vars – *this is critical, else you'll have license issues.*

Please set up your path and ld_library_path manually if the source_setup fails to locate your icc (Intel C compiler).

These must be sourced, not executed. Don't do

```
./source_setup  <-- WRONG!
```

If something doesn't work for you, come back and carefully repeat this step.

IV. Getting ready

Find and read the README.

```
cd YOURDIR/tutorial
less README
```

That will point you to where the documents are, in /usr/share/doc/hStreams.

hStreams_Overview.pdf - gives the big picture

hStreams_Tutorial.pdf - intro to how everything works, API

hStreams_Reference.pdf - API reference document

It also tells you what files you need to include, and which APIs you'll need.

V. Getting going

You'll progress through parts A, B and C, with to sub-parts 1, 2, … within each

```
cd YOURDIR/tutorial/A.remote_invocation
less README
cd 1.hello_hStreams_world
```

Look at the starter file

```
vim hello_hStreams_src.cpp
```

Look for //!!a and so on, as described in the README and the list above.

Modify that file as directed

Build with (make user also works)

```
make
```

Run with

```
./run.sh
```

Check against the solution file
Build with (make user also works)
`make solution`
Run with
`./solution_run.sh`

If you want to pass your own arguments to check runtime performance do.
./solution_run.sh args1 arg2 arg3 …

When you're done, move on to the next exercise
`cd ../2.pass_scalar_args_get_result`
and so on.
Explanation of the makefile:

I find it very convenient to use file completion, with A.<space> and 1.<space>