HSTREAMS TUTORIAL

March, 2016

Hetero Streams Library 1.0

THE NEW CENTER OF POSSIBILITY

Value of hStreams

- **Intel(R) hStreams is a library that supports task concurrency on heterogeneous platforms**
- **The concurrency may be**
  - Across nodes (Xeon, KNC)
  - Within a node for small matrix operations
  - In the overlapping of computation and communication, particularly for tiled solutions
- **hStreams relieves the user of complexity**
  - Pipelining
  - Thread affinitization
  - Asynchrony and offloading
  - Memory types
  - Memory affinitization

AN EASY ON RAMP TO MANAGE TASK CONCURRENCY ON HETERO PLATFORMS

Outline

Related documents

- **hStreams_Release_Notes.pdf**
  - Release notes
- **hStreams_Overview.pdf**
  - Description of the goals, objectives, design, performance and terminology for hStreams; this is a good place to start.
- **hStreams_Reference.pdf**
  - Programming Guide and API Reference
- **hStreams_Porting_Guide.pdf**
  - How to port from Intel® MPSS 3.6 to Hetero Streams Library 1.0

A PATH TO EASE OF USE

THE NEW CENTER OF POSSIBILITY

Ease of use

The majority of HPC developers want “easy” and selective control

- Easy on ramp, pay as you go – the more you describe, the more performance you get
- Don’t make them do much to get started
- Be transparent and give them as much control as they want

A path to ease of use

- Accommodate programmers of various skill levels
  - Scientist/novice, tuner/computer scientist, platform expert/ninja
- Intuitive and natural interfaces
  - Imperative sequence of commands
  - Declarative properties for memory

Separation of concerns

Majority of scientists – “don’t force me to become a computer scientist!”

- Express *what* the algorithm does – expose parallelism, option to declare properties and access patterns

Tuners – target new platforms

- Selectively exert control over *how* the work should be optimized
- May lack the application domain expertise that the original scientist-developer had

Ninjas – provide the best way

- Generate building blocks that make it progressively easier for tuners

With this separation, one can port to new platforms with a few localized changes

Natural, intuitive interfaces

Imperative interface for tasks

- Sequence of actions – compute tasks, transfer data, synchronize
- Dependences inferred from actions' operands
- Many users don't wish to provide a task graph – give them a way!

Declarative interface

- Describe operands in a pay as you go scheme
  - The more you describe, the greater the potential for concurrency
- Describe memory as buffers with optional properties

CONCEPTS

THE NEW CENTER OF POSSIBILITY

Concepts

- Key features
- Abstractions
- Hello world
- Streams
- Actions
- Heterogeneous platform
- Domains
- Buffers
- Queuing model and synchronization
- Invocation
- Matrix multiply example
- Sample sequence
- Review

Key features

Heterogeneous platforms

- Easy way to invoke, whether local or remote
- Fill a gap wrt OpenMP tasks by offering a uniform interface for local and remote
- Target whatever we need: Xeon Phi LB, Xeon and Xeon Phi SB, FPGAs, remote

Task concurrency

- Across different kinds of resources, within the same resource, pipeline comms & computes
- Seamlessly expand to fill available resources, when user programs tasks to use threads
- In order semantics, with out of order execution

Declarative memory interface

- Specify where to allocate, in what kind of memory, with what affinity, with which policies
- Expose dependences by describing operands or describing buffers

Abstractions

Domains are homogeneous subsets of resources in a hetero platform

- Memory coherence domains create a natural partition
- Tuners can further refine domains according to locality, e.g. KNL sub-NUMA clusters
- Uniform interface for tasks, regardless of domain
- Code can be optimized for each domain; targeting a domain implicitly selects the version

Streams are FIFO queues across which task parallelism is achieved

- Each domain can have its own stream, to enable task concurrency across hetero nodes
- Tuners can further subdivide domains into many streams to increase concurrency, efficiency
- Code mapped to stream expands to fill available resources, using OpenMP or TBB
- Number and width of streams is easily varied – easy for tuners to explore a search space

Buffers represent program variables in a system-wide, unified address space

- Buffers can be instantiated on just the subset of domains on which they are needed
- Application developers can specify access properties
- Tuners can specify memory type, pinning, aliasing, affinity, policies
- Dependences are inferred based on operands that fall within buffers

Example: hStreams Hello World

source

```
// Main header for app API (source)
#include <hStreams_app_api.h>
int main() {
        uint64_t arg = 3735928559;
        // Create domains and streams
        hStreams_app_init(1,1);
        // Enqueue a computation in stream 0
        hStreams_app_invoke(0, "hello_world",
                1, 0, &arg, NULL, NULL, 0);
        // Finalize the library. Implicitly
        // waits for the completion of
        // enqueued actions
        hStreams_app_fini();
        return 0;
}
```

sink

```
// Main header for sink API
#include <hStreams_sink.h>
// for printf()
#include <stdio.h>
// Ensure proper name mangling and symbol
// visibility of the user function to be
// invoked on the sink.
HSTREAMS_EXPORT
void hello_world(uint64_t arg)
{
        // This printf will be visible
        // on the host. arg will have
        // the value assigned on the source
        printf("Hello world, %x\n", arg);
}
```

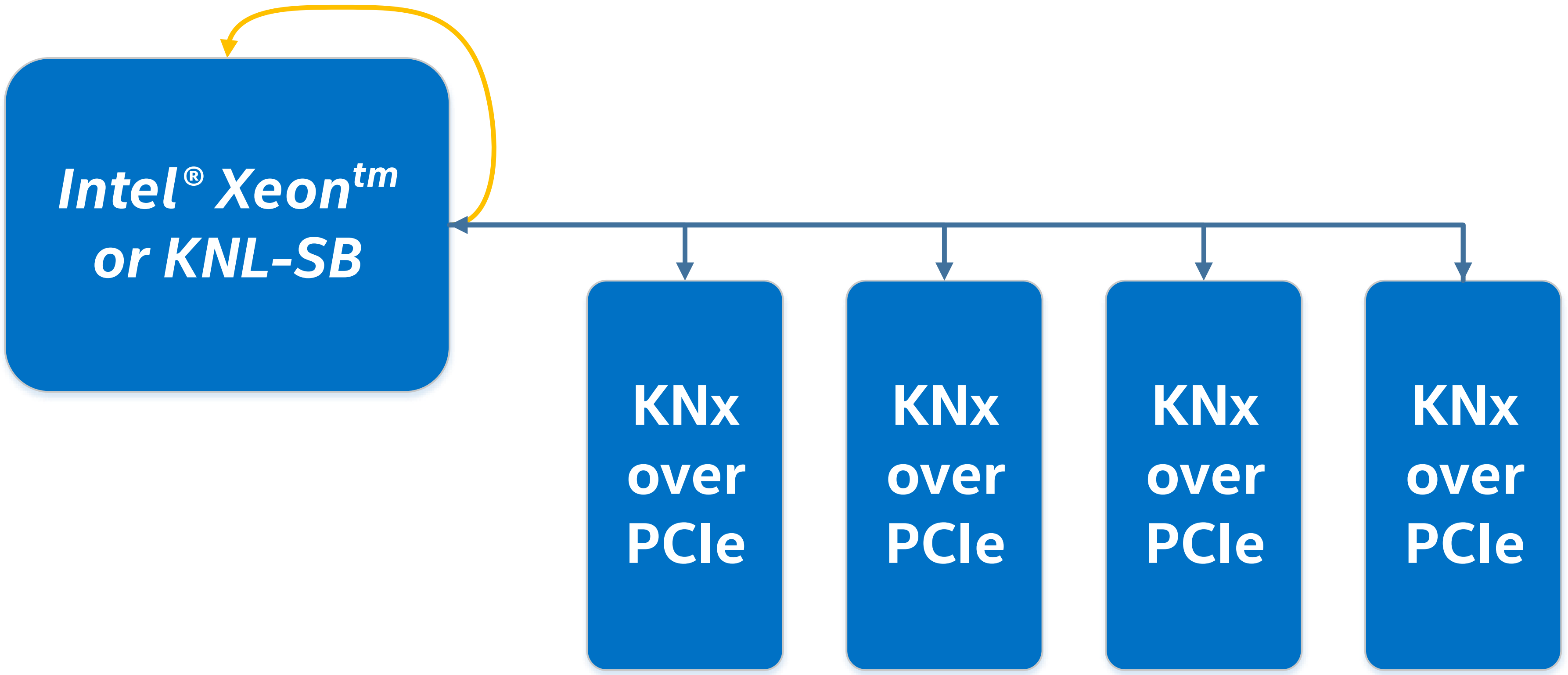
Streams

- **A <u>stream</u> is a queue with sequential (FIFO) semantics**
- **Actions are enqueued in the stream**
- **The stream (queue) has two endpoints**
  - The head is the source, from which actions are issued
  - The tail is the sink, at which actions are (logically) executed
  - These two endpoints are logically distinct (unlike OpenMP tasks)
- **The sink is bound to a set of resources**
  - Subset of threads on a given target (e.g. card, host), which may have its own memory coherence domain

Actions

- **A subset of the APIs are actions**
  - Actions are enqueued in a stream, and have a stream ID
  - All actions are non-blocking
  - Since actions are asynchronous, they have a completion event
  - Non-action APIs are blocking, and execute in the source thread
- **There are three kinds of actions**
  - **Compute** – remote invocation
  - **Data transfers** – to or from at least one of the stream's endpoints
  - **Synchronization** – with the source or within the stream

Compute resources: reach the hetero platform



- 3.4: KNC, over PCIe – like NVidia/CUDA Streams* and AMD/OpenCL*
- 3.6: Xeon, stream to self - differentiated from competition

**Some trademarks are the property of others. KNC and KNL are members of the Intel® Xeon Phi™ Coprocessor family.*

Domains

- **A domain is a set of resources**
  - Compute: device and a mask indicating a subset of its CPUs
  - Storage: a distinct memory space
- **Scope limits**
  - At most one OS instance (can be multiple sockets)
  - At most one memory coherence domain (can be multiple NUMA domains)
  - Single kind of computational capability within a domain, but different domains can be bound to different kinds of compilation targets, e.g. host and card
  - Example: Memory coherence domain and its resources
  - Example: NUMA subdomain and its nearest resources

Buffers and dependences

Architectural restrictions

- Buffers are 1D

Current limitations, until we add operand descriptors

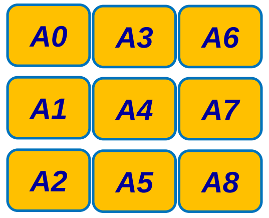
- Dependences will be enforced at the buffer granularity
- Transfer operands are 1D

Implications, for now, if you want a 2D tile to be the dependence granularity

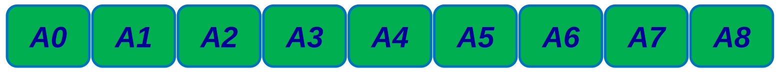
- Marshall your own data by copying from 2D array into 1D temp arrays
- Wrap the dependence granularity as a buffer: 1D temp arrays instead of original 2D variable

```
hStreams_app_create_buf(HostProxyAdr=TempArrayA[0], NumBytes=4096*1024)
…
hStreams_app_create_buf(HostProxyAdr=TempArrayA[26], NumBytes=4096*1024)
```

- Use those 1D temp arrays as operands for copies to the sink, and task operands



2D array, A



1D arrays, A0-A8

Buffer instantiation

- **Buffers are instantiated per domain, not per stream**

  - Example: all or subsets of tiles of array could be instantiated on any subset of Host & MICs
  - Each buffer only needs to be transferred between Host and MIC0 or MIC1 once per domain

Cross-stream synchronization

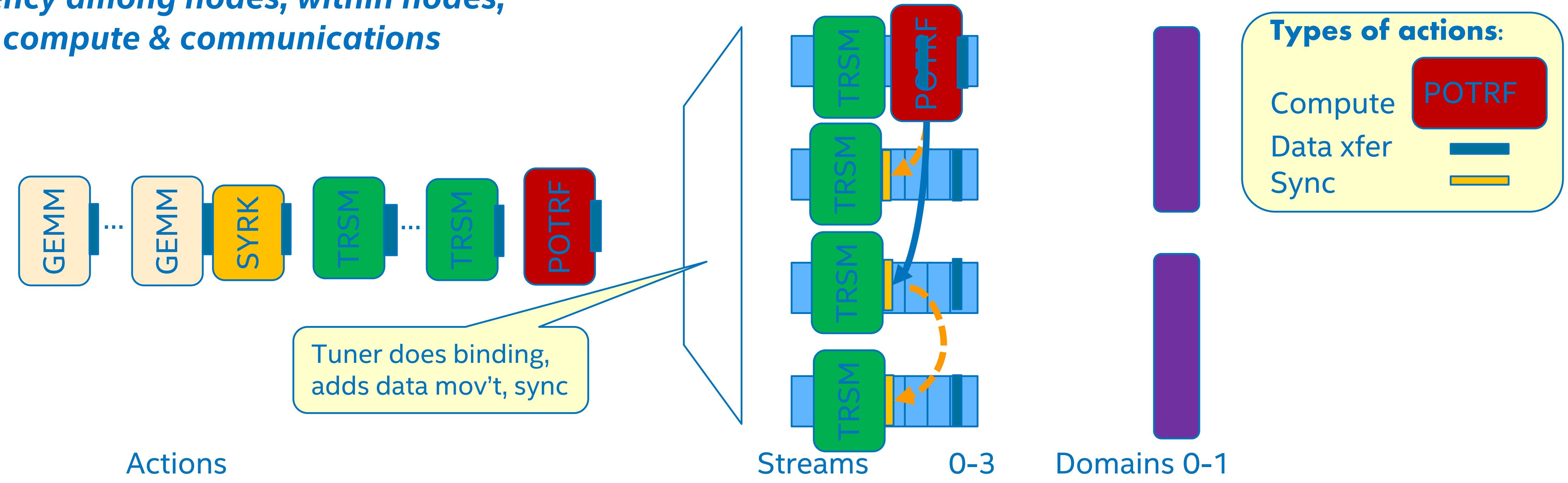
Cross-stream synchronization

```
• hStreams_app_invoke(LogStrID=0,"POTRF",…, Event=&Event1)
• hStreams_app_event_wait_in_stream(LogStrID=1, &Event1, SrcAdr )
• hStreams_app_xfer_memory(LogStrID=0,SrcAdr=A[5],DestAdr=B[3],
  NumBytes=4096,XferDirection=HSTR_SRC_TO_SINK, &Event2)
• hStreams_app_event_wait_in_stream(LogStrID=3, &Event2, DestAdr )
```

Completion event

Deps induced on only DestAdr

Concurrency among nodes, within nodes, between compute & communications



Invocation (until we have a nifty C++ template function for this)

Source side

User function, bloo, which makes a call to

User-coded wrapper function call_remote_bloo, which marshals args to 64b scalar and heap args, and then calls…

hStreams_EnqueueCompute, which manages dependences, maps args into a sized memory blob, performs address translation for heap args, and calls

lower-level plumbing function: wrapper that does remote invocation, using COIPipelineRunFunction, with a sized memory blob

Sink side

lower-level plumbing function: thunk, called by COI, that receives the data blob and unmarshals that to 64b scalar and heap args, and then calls…

user-coded thunk function remote_thunk_bloo, which unmarshals 64b scalar and heap args, and then calls…

user-coded or standard library native function bloo

Unified task interface for a heterogeneous platform

- Tasks associated with those 9 tiles can be scheduled on any of the (9) streams
- The invocation interface is the same, regardless of the target type

- The scientist-programmer can expose tasks
- The tuner can do the binding of tasks to streams, potentially favoring one type of target over another for a given task type, and potentially load balancing across the streams.

- For now, binding and scheduling is fully manual; in the future, it will become more automated and dynamic

Example: matrix multiply

Domains

- 28-core Haswell/Xeon host, 61-core 7120P mic card 0, 57-core 3110P mic card 1
- User indicates which resources to use via environment variable; default is local MIC cards

Streams: 9-way concurrency across 3 domains

- Haswell: reserve 1 core, create 3 streams (28-1)/3=9 cores each
- MIC0: reserve 4 threads, create 3 streams (244-4)/3=80 threads each
- MIC1: reserve 4 threads, create 3 streams (228-4)/3=74 threads each

Buffers – 2 options for managing dependences

- Fine granularity, with no operand descriptors vs. Coarse granularity, with operand descr
  - AxB=C, with 3x3 2D tiles for each
  - Copy into 3 (for A,B,C) x 9 (for 3x3) 1D buffers vs. Wrap original user memory for A,B,C in 3 buffers, and let implementation is free to marshal the data in most-efficient way
  - Refer to the 27 1D buffers at function calls vs. use operand descriptors at function calls to describe 2D tiles
  - Instantiate those buffers only where they are needed, on the host, MIC0, MIC1

Sample sequence

Initialize

```
hStreams_app_init(StreamsPerDomain=4,…)
```

Allocate buffers

```
hStreams_app_create_buf(HostProxyAdr=ArrayA, NumBytes=4096*1024)
```

Transfer memory there

```
hStreams_app_xfer_memory(LogStrID=3,SrcAdr=A[5],DestAdr=B[3],
NumBytes=4096,XferDirection=HSTR_SRC_TO_SINK, Event=&Event1)
```

Remote invocation

```
hStreams_app_invoke(LogStrID=3,"myfunc",ScalarArgs=2,HeapArgs=3,
  ArgArray=Args,ReturnVal=NULL,RetValSize=0, Event=&Event2)
hStreams_app_dgemm(Order,TransA,TransB,M,N,K,…)
```

Transfer memory back

```
hStreams_app_xfer_memory(LogStrID=3,SrcAdr=C[7],DestAdr=C[7],
NumBytes=4096,XferDirection=HSTR_SINK_TO_SRC, Event=&Event3)
```

Synchronize

```
hStreams_app_thread_sync()
hStreams_app_stream_sync(LogStrID=3)
```

Finalize

```
hStreams_app_fini()
```

Review of hierarchy

- **Physical resources to which streams are mapped**
  - Memory coherence domain and its resource; reflects HW organization
  - By default, the set of visible KNxes over PCI; specifiable in the future
  - We call those physical domains
- **Map logical to physical domains**
  - Potentially a many to one relationship
    - *Can be used to narrow scope, e.g. to NUMA subdomain, memory type*
    - *Can be used to port from many physical domains to fewer*
    - *Separate topic: how this helps portability*
  - Each logical domain has a CPU affinity mask
    - *Logical domains are disjoint, and do not span devices*
- **Streams are bound to logical domains**
  - Each stream has a CPU affinity mask within a single logical domain
    - *A league is a set of disjoint streams that cover all CPUs in a log domain*
    - *Separate topic: overlapping sets of streams*
- **Map logical streams to physical streams**
  - Potentially a many to one relationship
    - *Oversubscription leads to interleaving; just a convenience*

API OVERVIEW

THE NEW CENTER OF POSSIBILITY

API overview

- This covers the most common app APIs

- Configuration
- Buffers
- Data transfers
- Invocation
- Synchronization

APIs: keep it simple but flexible

- **Most users won't need full richness of hStreams**
- **So keep things simple with an "app API" layer**
  - Self contained for most uses, but may be mixed with core APIs
  - Fewest-possible arguments
  - Calls lower "hStreams core" layer
  - Precompiled into distributed hStreams library for ease of use
- **But flexible**
  - App API source code is available
  - Can copy it, rename functions, specialize, recompile
  - Build your own library, with incremental changes
  - Completely encapsulates lower layers
- **For more details on APIs, see**
  - This document – tutorial overview of APIs and reference code
  - hStreams_Reference.pdf – Programming Guide and API Reference

Configuration

- **Configuration options**
  - High-level (app API), simplified, at initialization
  - Low-level (core API), full control, at initialization
  - Low-level full control, as you go along
- **Initialization with app API**
  - Creates {physical,logical}x{domains,streams} with architected numbering
  - These assume 1 logical domain per physical domain
  - User can set StreamsPerDomain according to available task parallelism & task efficiency (e.g. more streams for small matrices)
  - LogStreamOversubscription is usually 1
  - Second init API for enables streams per domain to vary, e.g. for load balancing

```
HSTR_RESULT
hStreams_app_init         (uint32_t     in_pStreamsPerDomain,
                           uint32_t     in_LogStreamOversubscription);
HSTR_RESULT
hStreams_app_init_domains(uint32_t      in_NumLogDomains,
                           uint32_t    *in_pStreamsPerDomain,
                           uint32_t     in_LogStreamOversubscription);
```

Configuration usage notes

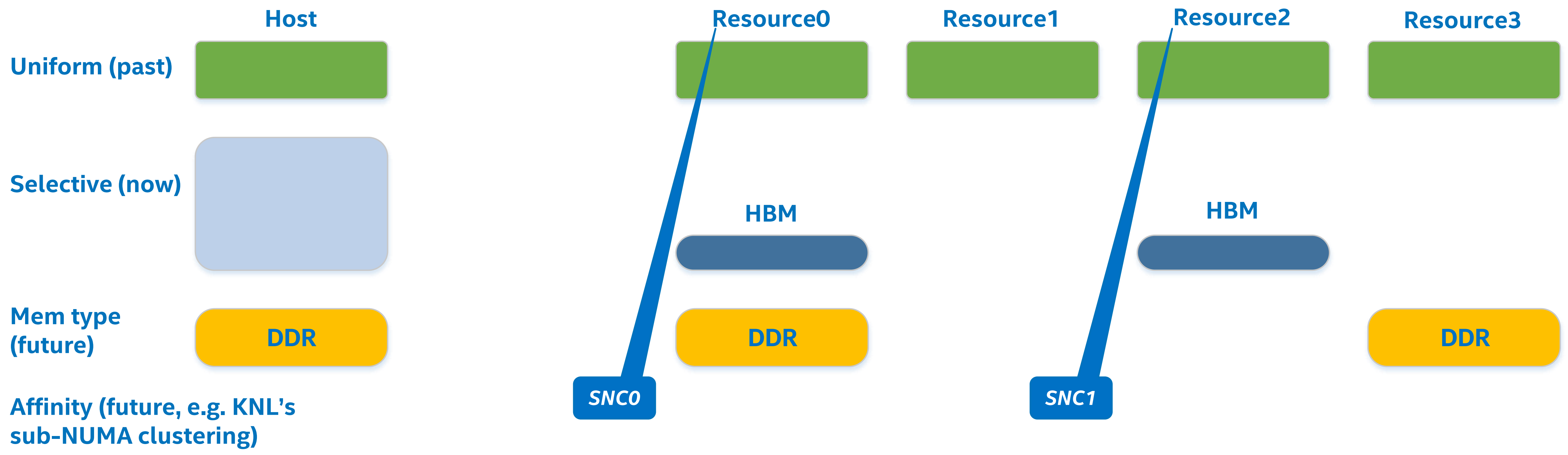
- **Creating domains**
  - User doesn’t have to enumerate and accommodate # devices
    - *Full control if they want it*
- **Creating N streams per domain**
  - User doesn’t have to enumerate and accommodate CPUs/chip
    - *Same for 5110 and 7120*
- **Sets of streams can be incrementally added**
  - If # streams in a given logical domain is 0, that will be skipped
  - # streams for a given logical domain may vary across calls to create multiple leagues, e.g. one with 3 streams and another with 4
- **Host support will grow in forthcoming release**
  - Create streams manually for host, not added to app APIs yet
  - See host_multicard versions of ref codes for Cholesky and mat mult
  - Windows support is not added yet

Buffers

- **Buffers encapsulate memory that is recognized and managed by hStreams**
  - Required for dependence analysis
  - Unit of allocation
  - Can have properties, e.g. memory type, pinned, affinitized, aliased
- **Memory allocation can be done by**
  - User, then wrapped with a call to hStreams
  - hStreams, if in some other domain
    - *hStreams causes allocation to adhere to specified properties*

```
HSTR_RESULT
hStreams_app_create_buf(void           *in_BufAddr,
                        const uint64_t in_NumBytes);
HSTR_RESULT
hStreams_AllocEx(void                *in_BufAddr,
                 const uint64_t      in_NumBytes,
                 HSTR_BUFFER_PROPS *in_pBufferProps,
                 int64_t             in_NumLogDomains,
                 HSTR_LOG_DOM      *in_pLogDomainIDs);
```

Managing memory: make it easier to use future IA features



- **Uniform** – always instantiate buffers across all resources
- **Selective** – only instantiate where needed
- **Memory type** – optionally specify type, make it happen
- **Affinitized** – optionally cause 1st touch from affinitized threads
    - *Not supported yet*

Buffer usage notes

- **Buffers referenced with source domain addresses**
  - This is all that the user, invoking from the source domain, can see
- **Shape**
  - Allocation, pinning is expected to be only on 1D arrays, not MultiD
- **Properties**
  - Can name the type, e.g. HBM, and not have to learn all APIs
  - Can do remote allocation
  - Can specify affinity, and have runtime do all the work for 1st touch
- **Future topics**
  - Pinning
  - Consistency of properties across all instances

Buffers: logical and physical

- **Buffers instantiation**
  - There is a single logical buffer
  - Physical instantiation is selectable for `app_init_domains`
    - *Example: Big buffer only on host, small working buffers on device*
  - Physical instantiation occurs in the source and all domains for `app_init`
- **The hStreams implementation maps addresses**
  - All references to memory in remote invocations must use heap arguments; no globals allowed. Statics are permitted.
  - At invocation in a given domain, heap arguments are transparently mapped to addresses in that domain
  - User data structures that contain pointers will not get fixed up – this means that C++ may involve lots of manual user intervention

Dependence management

- **Streams provide a FIFO abstraction**
  - Compute, communication and synchronization actions enqueued in stream
  - Architecturally processed in FIFO order
  - Implementation may process out of order, subject to dependence checking
- **Dependences are managed at buffer granularity**
  - Each heap address in a data transfer or remote invocation is mapped to a buffer
  - Dependences are tracked per stream, at buffer granularity
  - If no dependence, DMAs can complete out of order with respect to each other and with respect to computes. Computes that use the same resources (same stream) complete in order.
  - Support for smaller granularities is not yet available
- **Completion events**
  - Can be used to enforced dependences by operations in source thread or another stream

Granularity

- **Why make buffers bigger?**
    - Encapsulate all the data that needs to be moved at a time
    - Simpler
    - Less per-buffer overhead
- **Why make buffers smaller?**
    - Enable greater concurrency among buffers, since per-buffer operations are sequentialized within streams (enforced by implementation) and across streams (user responsibility)

Data transfers

- **Explicitly scheduled**
- **Endpoints**
  - Simple: source to sink or sink to source
  - General: one of source or sink can be in another domain
- **Shape**
  - For now, only 1D supported

```
hStreams_app_xfer_memory(HSTR_LOG_STR LogStrID,
                         void* WriteAdr, void* ReadAdr,
                         uint64_t NumBytes,
                         HSTR_XFER_DIRECTION XferDirection,
                         HSTR_EVENT Event)
hStreams_EnqeueueDataXDomain(HSTR_LOG_STR LogStrID,
                         void* WriteAdr, void* ReadAdr,
                         uint64_t NumBytes,
                         HSTR_LOG_DOM SrcLogDom, HSTR_LOG_DOM DestLogDom,
                         HSTR_EVENT Event)
```

Data transfer usage notes

- **Addressing**
  - Unified global address space, using source proxy virtual addresses
  - To reference a specific instance of a buffer, give its global address and that instance
- **Source and destination**
  - Can be in different buffers or same buffer
  - Can be any address or size, but can’t extend past buffer bounds
  - Source and dest can’t be overlapping if same instance
  - At least one of source or sink domain must be an stream endpoint
- **Optimization**
  - All transfers between logical domains must be explicit
  - If two logical domains are mapped to the same physical domain and the buffer is explicitly enabled for aliasing, then the act of transferring can be optimized away; dependences still maintained

Invocation

- **Can use provided convenience functions**
  - *gemm, memcpy, memset
  - More available as reference functions, e.g. POTRF, TRSM,SYRK
- **Can create user functions**
  - Natural form of API called at source and sink
  - Must write wrapper and thunk to marshall and unmarshal

```
hStreams_app_invoke(HSTR_LOG_STR LogStrID,
                    const char *in_pFuncName,
                    uint32_t    ScalarArgs, uint32_t HeapArgs,
                    uint64_t   *in_pArgs,
                    HSTR_EVENT *out_pEvent,
                    void       *out_pReturnVal,
                    uint32_t    RetValSize)
hStreams_app_dgemm(LogStrID,
                   Order,TransA,TransB,M,N,K,…, out_pEvent)
```

Invocation usage notes

- **Target functions can't be outlined, as for compiler**
- **Target function must be visible in a dynamic lib**
  - Compile for each target, target type is part of lib name
  - Library name implicitly loaded if follows a convention
    - *<exe_name>_mic.so is on SINK_LD_LIBRARY_PATH*
    - *<exe_name>_host.so is on HOST_SINK_LD_LIBRARY_PATH*
  - Library name can be explicitly specified instead
  - Loading happens at init time

Synchronization

- **Can be blocking**
  - Granularities are all streams associated with source thread, single stream, or a set of specific completion events

```
hStreams_app_thread_sync()
hStreams_app_stream_sync(HSTR_LOG_STR LogStrID)
hStreams_app_event_wait(uint32_t    in_NumEvents,
                        HSTR_EVENT *in_pEvents)
```

- **Or non-blocking, inserted into a stream**
  - Can wait on zero or more events
  - Can induce waits for a limited set of buffers

```
hStreams_app_event_wait_in_stream(
    HSTR_LOG_STR       in_LogStreamID,
    uint32_t           in_NumEvents,
    HSTR_EVENT        *in_pEvents,
    int32_t            in_NumAddresses,
    void             **in_pAddresses,
    HSTR_EVENT        *out_pEvent);
```

Synchronization usage notes

- **Examples of supported usages**
  - Blocking calls
    - *Immediately wait on completion event*
  - Timing
    - *Start timing when a previously-scheduled action starts*
    - *End timing when a previously-scheduled action completes*
    - *End timing when all previously-scheduled actions in stream complete*
  - Actions in stream B that use operand X can be made to wait on an action in stream A that produces X
- **Ease of use**
  - Users (e.g. **Barcelona Supercomputing Center**) list many ways that hStreams sync design is better than that of CUDA Streams*

**Some names and brands may be claimed as the property of others*

REFERENCE CODES

THE NEW CENTER OF POSSIBILITY

Reference codes

- Preliminaries
- Get started with a simple performance run
- Matrix multiply example
- Test application
- More to come

To experiment with reference codes:

- **Set up compiler and its environment variables:**
  - . /opt/intel/composerxe/bin/compilervars.sh intel64 (in bash)
    - *This also sets MKLROOT environment variable.*
- **Copy installed reference code directory locally: e.g.**
  - cp –r /usr/share/doc/hStreams ~/myhStreams/
  - cd ~/myhStreams/ref_code/matMult (or basic_perf, test_app, …)
  - make
  - bash ./run*.sh
- **Set up environment variables**
  - ~/myhStreams/ref_code/common/setEnv.sh can be sourced to set the necessary environment variables
  - LD_LIBRARY_PATH, SINK_LD_LIBRARY_PATH, HOST_SINK_LD_LIBRARY_PATH
  - MKL_MIC_MAX_MEMORY
  - MIC_USE_2MB_BUFFERS
  - The same set of environment variables is used for all example reference codes.

LD_LIBRARY_PATH

- **export LD_LIBRARY_PATH=**
  - $LD_LIBRARY_PATH:/usr/lib64
- **The host /usr/lib64 directory contains**
  - libhstreams_source.so

SINK_LD_LIBRARY_PATH

- **export MPSS_LIB64=**
  - */opt/mpss/3.?*/sysroots/k1om-mpss-linux/usr/lib64/*
- **export SINK_LD_LIBRARY_PATH=**
  - *$MKLROOT/lib/mic/:*
  - *$MKLROOT/../compiler/lib/mic:*
  - *$MPSS_LIB64:*
  - *../../bin/dev*
  - The Intel® Math Kernel (Intel® MKL) libraries that come with your Intel® compiler include libs for the MIC device in *$MKLROOT/lib/mic*
  - The general libraries that come with your Intel® compiler include the OpenMP* library for the MIC device in the *$MKLROOT/../compiler/lib/mic* directory (**libiomp5.so** for MIC).
  - The device-side shared object that you create as part of your hStreams application is located in our examples in the **../../bin/dev** directory

HOST_SINK_LD_LIBRARY_PATH

- **The host-side shared object that you create as part of your hStreams application is located in our examples in the ../../bin/host directory**
    - export HOST_SINK_LD_LIBRARY_PATH=
        - *../../bin/host*
- **The standard MKL and compiler binaries are already found on the LD_LIBRARY_PATH**

Getting started with basic_perf

- **basic_perf is a simple, illustrative mini app**
- **Make the binary**
    - cd ~/myhStreams/ref_code/basic_perf
    - Modify driver parameters, e.g. SIZE, in basic_perf.cpp
    - make
- **Run it**
    - bash ./run_basic_perf.sh
- **This will show transfer performance and concurrent remote execution, for 1 and 4 streams**
    - Concurrent transfers don’t have much overhead
    - Computes have some benefit from concurrency
- **basic_perf uses high level app_api’s**

Block Matrix Multiplication

- **Tutorial instruction on how to run matMult**
  - cd ~/myhStreams/ref_code/matMult
  - make
- **Simplest invocation, with run_matMult**
  - bash ./run_matMult.sh
- **Instructions on more general invocation**
  - ../../bin/host/matMult –b500 –m2000 –n3000 –k4000
- **matMult uses hStreams app_API functions**
  - Initializes 4 logical streams on 4 partitions of a single MIC device
  - Splits up [A] & [B] matrices into blocks of size set by –b xxxx arg
  - Sends blocks of [A] and [B] to the MIC device, enqueues compute requests to Intel® MKL GEMM procedures
  - Copies blocks of [C] back to host from MIC device.
  - Assembles blocks of A,B,C and re-tests result on host
  - Checks for correct results.

Test Application

- **Make the binary**
  - cd ~/myhStreams/ref_code/test_app
  - make
- **Run it**
  - bash ./run_test_app.sh
- **Show all the options via help**
  - ./run_test_app.sh –h
- **Use 64k buffer and 8 partitions of MIC device**
  - bash ./run_test_app.sh -s 65536 -p 8
- **test_app uses the hStreams “Core” api’s**

IO Application

- **Make the binary**
  - cd ~/myhStreams/ref_code/io_perf
  - make
- **Run it**
  - bash ./run_io_perf.sh
- **Show all the options via help**
  - bash ./run_io_perf.sh –h

Cholesky Matrix Decomposition (L*L^t)

- **These reference codes compute the Cholesky decomposition of a symmetric matrix**
- **Let's do something useful:**
  - cd ~/myhStreams/ref_code/cholesky/
  - more README.txt
  - cd <subdirectory>
  - make
  - bash ./run_<subdirectory>.sh
- **Where <subdirectory> is one of:**
  - **tiled_host – just host native, without hStreams**
  - **tiled_hstreams – use hStreams for simple offload case**
  - **tiled_host_multicard – combo of host, cards using hStreams**
- **These use custom kernels for sinks in ~/myhStreams/ref_code/common**

A=LU Matrix Decomposition

- **This reference code computes the LU decomposition of an unsymmetric matrix.**
- **Let's do something else useful:**
  - cd ~/myhStreams/ref_code/lu/
  - more README.txt
  - cd <subdirectory>
  - make
  - bash ./runit.sh
- **Where <subdirectory> is one of:**
  - tiled_host – just host native, without hStreams
  - tiled_hstreams – use hStreams for simple offload case
- **Shows the use of custom kernels for sinks in the /common/ folder.**

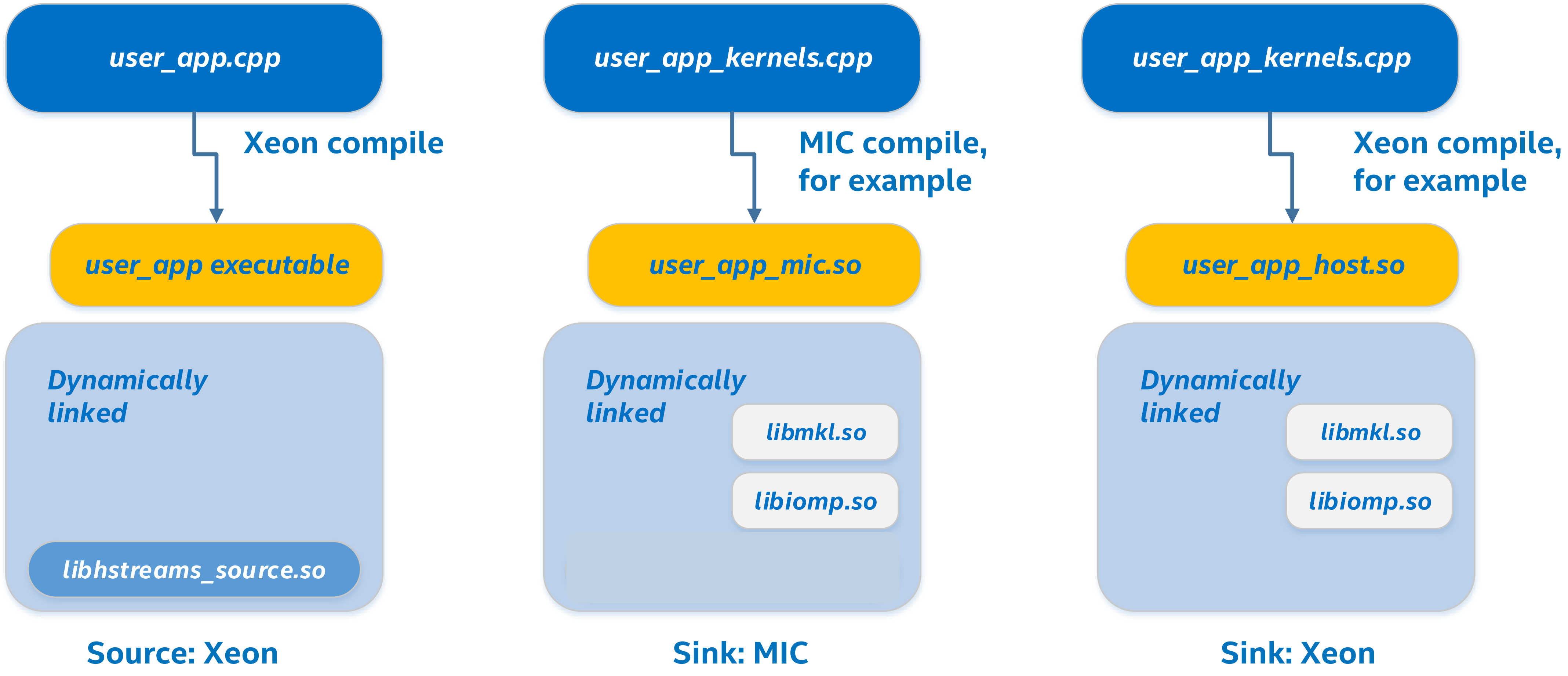
HOW THE LIBRARY WORKS

THE NEW CENTER OF POSSIBILITY

How the library works

- **User code and libraries**
- **Invocation**
    - Code at the caller and callee
    - What the wrapper and thunk do
- **Creating your application**

User code and libraries



- Users provide their source and sink-side app code
- Each of the source and the sink for MIC are compiled in separate modules
- The sink-side code for using the host as a target *may* be a separate module

User code and libraries (cont'd)

- **Sink-side user_app libraries are optional.**
- **All card-side user_app libraries must be located in the $SINK_LD_LIBRARY_PATH collection of directories**
- **When the host is a target (sink), all host-sink-side user_app libraries must be located in the $HOST_SINK_LD_LIBRARY_PATH collection of directories**
- **By default, hStreams initialization attempts to load**
  - a card-side library with the host side executable name with '_mic.so' suffix
  - a host-sink-side library with the host side executable name with '_host.so' suffix
- **hStreams supports optionally explicitly naming card-side libraries to be loaded during hStreams initialization via the hStreams options mechanism**
  - See below for details

Explicitly setting lib names to load

```
HSTR_OPTIONS hstreams_options;
    hStreams_GetCurrentOptions(&hstreams_options,
         sizeof(HSTR_OPTIONS));

    hstreams_options.verbose = 0;
    char *libNames[200] = {NULL, NULL};
    char *libNamesHost[200] = {NULL, NULL};

    // Library to be loaded for sink-side code
    libNames[0] = "my_app_sink_1.so";
    hstreams_options.libNameCnt = 1;
    hstreams_options.libNames = libNames;

    // Library to be loaded for host-sink-side code
    libNames[0] = "my_app_host_1.so";
    hstreams_options.libNameCntHost = 1;
    hstreams_options.libNamesHost = libNamesHost;
```

Invocation

- Parameters of target function
  - Scalar args, then heap args
  - All must be 64 bits. Otherwise bits will get incorrectly mapped into arguments
  - Caller and callee outside of hStreams are responsible for type casting
  - Return value can have variable size
- Be sure to type cast carefully
- Possible user contributions
  - C++ template-based wrappers
  - String format convenience wrappers

What the “wrapper” and “thunk” do

- **There’s a “wrapper” at the source and a “thunk” at each sink**
- **Source “wrapper”**
  - Marshalls arguments into a buffer
  - Maps named functions to remote function addresses
  - Asynchronously queues up work
  - Yields a synchronization object handle
- **Sink “thunk”**
  - De-marshalls arguments, calls named sink-side function
- **hStreams**
  - Map heap addresses from source proxy address to sink-side domain’s address
  - Performs remote invocation, including passing the arguments

Invocation example

SOURCE SIDE (e.g. host)

```
 // Prep arguments
   uint64_t arg0, arg1, arg2, arg3, arg4;
   char* buf_adr_in = ...;
   char* buf_adr_out = ...;
   arg0 = 0xaaaa;
   arg1 = OTHER_LEN;
   arg2 = TEST_BUF_SIZE;
   uint64_t args[5];
   args[0] = (uint64_t)arg0;
   args[1] = (uint64_t)arg1;
   args[2] = (uint64_t)arg2;
   args[3] = (uint64_t)(buf_adr_in);
   args[4] = (uint64_t)(buf_adr_out);

   // Enqueue compute
CHECK_HSTR_RESULT(
hStreams_EnqueueCompute(
   0,                      // logical stream 0
  "test_func_s3h2",        // in_pFunctionName
   3,                      // in_numScalarArgs
   2,                      // in_numHeapArgs
   (uint64_t*)(args),      // in_pArgs
   NULL,                   // Event handle
   NULL,                   // out_pReturnVal
   0));                    // in_ReturnValueSize
```

Args can only be uint64_t

Have to explicitly differentiate scalar and heap args

See test_app.cpp and offloaded_function_sink.cpp for an example of return value usage

SINK SIDE (e.g. MIC)

```
COINATIVELIBEXPORT
void test_func_s3h2(
                uint64_t arg0,
                uint64_t arg1,
                uint64_t arg2,
                uint64_t bufi,
                uint64_t bufo) {
 int i;
 char *buf_in = (char*)bufi;
 char *buf_out = (char*)bufo;

 printf("Arrived in test_func_s3h2
   with arg0 %lx, arg1 %lx, arg2 %lx,
   buf_in %lx and buf_out %lx.\n",
   arg0, arg1, arg2, buf_in, buf_out);

 // buffer copy
 memcpy(buf_out, buf_in, arg2);
 // overwrite with another value
 for (i = 0; i < arg1; i++)
   buf_out[i] = arg0;
 }
}
```

Scalar, then heap args

Must be void

No global vars all heap refs are wrt args

Host-side invocation and alloc

```
int main() {
  uint32_t streams_per_logdomain = MAX_CONCURRENCY;
  uint32_t log_stream_oversubscription = 1; // keep it simple
  HSTR_EVENT events[MAX_CONCURRENCY];
  uint32_t stream;
  double timeBegin, timeEnd;
  int iters;
  // DATA_TYPE is set above
  // MAX_CONCURRENCY enables working with many of these at a time
  DATA_TYPE **A[MAX_CONCURRENCY],
            **B[MAX_CONCURRENCY],
            **C[MAX_CONCURRENCY];
  int conc;

  dtimeInit();
  // This is non-performant, but easier to read
  for (conc = 0; conc < MAX_CONCURRENCY; conc++) {
    A[conc] = (DATA_TYPE **)malloc(sizeof(DATA_TYPE)*SIZE*SIZE);
    B[conc] = (DATA_TYPE **)malloc(sizeof(DATA_TYPE)*SIZE*SIZE);
    C[conc] = (DATA_TYPE **)malloc(sizeof(DATA_TYPE)*SIZE*SIZE);
  }
```

User allocates host memory

Init and create buffers

```
// Iterate through number of streams_per_logdomain: 1, MAX_CONCURRENCY
  for (streams_per_logdomain = 1; streams_per_logdomain <= MAX_CONCURRENCY;
    streams_per_logdomain *= MAX_CONCURRENCY) {

    //      init
    printf(">>init\n");
    CHECK_HSTR_RESULT(
     hStreams_app_init(
       streams_per_logdomain,
       log_stream_oversubscription));

    //      create_bufs
    printf(">>create bufs\n");

    // Walk through streams
    for (stream = 0; stream < streams_per_logdomain; stream++) {

      CHECK_HSTR_RESULT(
       hStreams_app_create_buf(A[stream], sizeof(DATA_TYPE)*SIZE*SIZE));
      CHECK_HSTR_RESULT(
       hStreams_app_create_buf(B[stream], sizeof(DATA_TYPE)*SIZE*SIZE));
      CHECK_HSTR_RESULT(
       hStreams_app_create_buf(C[stream], sizeof(DATA_TYPE)*SIZE*SIZE));
    }
```

Domain: card
Vary # streams
in domain

Initialize:
streams_per_logdomain – streams per domain
log_stream_oversubscription - oversubscription

Create buffers
independent of streams

Data transfer

```
// Assume writes to buffers already occurred
    //------     Begin xfer_memory TO domain     -------------------
    printf(">>xfer mem to remote domain\n");

    // Begin timing
    timeBegin = dtimeGet();

    // Walk through timing iterations
    for (iters = 0; iters < ITERATIONS; iters++) {

      // Walk through streams in a single domain
      for (stream = 0; stream < streams_per_logdomain; stream++) {

             // Transfer to remote domain
             CHECK_HSTR_RESULT(
                 hStreams_app_xfer_memory(A[stream], A[stream], // src & dest addr, as though in source domain
                                          sizeof(DATA_TYPE)*SIZE*SIZE,
                                          stream,            // Logical stream
                                          HSTR_SRC_TO_SINK,  // Xfer_Direction
                                          NULL));            // completion event; wait for thread vs. indiv xfer
             CHECK_HSTR_RESULT(
                 hStreams_app_xfer_memory(B[stream], B[stream], // src & dest addr, as though in source domain
                                          sizeof(DATA_TYPE)*SIZE*SIZE,
                                          stream,            // Logical stream
                                          HSTR_SRC_TO_SINK,  // Xfer_Direction
                                          NULL));            // completion event; wait for thread vs. indiv xfer
      }
    }

    //      thread_sync just once, vs. every iteration
    CHECK_HSTR_RESULT(
     hStreams_app_thread_sync());

    // End timing
    timeEnd = dtimeGet();
```

Transfer input buffers explicitly

Assure completion before timing

DGEMM

```
//------     Begin xGEMMs in remote domain     -------------------
   printf(">>xgemm - sample fixed functionality\n");
   double alpha = 1.0;
   double beta  = 1.0;

   // Begin timing
   timeBegin = dtimeGet();

   // Walk through timing iterations
   for (iters = 0; iters < ITERATIONS; iters++) {

     // Walk through streams in a single domain
     for (stream = 0; stream < streams_per_logdomain; stream++) {

       // Nothing special about use of macros for DATA_TYPE and HSTREAMS_APP_XGEMM
       //  These are just used here so it's easier to change types and function names
       CHECK_HSTR_RESULT(
        hStreams_app_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                     SIZE, SIZE, SIZE,  // M, N, K - square in this case
                     alpha,
                     (DATA_TYPE*)A[stream], SIZE,
                     (DATA_TYPE*)B[stream], SIZE, beta,
                     (DATA_TYPE*)C[stream], SIZE,
                     stream,
                     NULL));  // completion event; wait for thread vs. indiv computes

     } // for all streams
   } // for all iterations

   //     thread_sync just once, vs. every iteration
   CHECK_HSTR_RESULT(
    hStreams_app_thread_sync());
```

Remote invocation
Use stock primitive for DGEMM

Creating your application

- **What the user is responsible for**
  - Calling hStreams APIs
    - *Initialize and create streams*
    - *Create buffers*
    - *Remote invocation*
    - *Synchronization*
    - *Finalize*
  - Building application
  - Writing sink-side functions if you wish to use more than the stock convenience functions (mem* and *gemm)

Building and running

- **Include header**
  - #include <hStreams_app_api.h> for only high-level abstraction, with #include <hStreams_app_api_sink.h> on sink side
  - #include <hStreams_source.h> for hStreams core usage, with #include <hStreams_sink.h> on sink side
- **User-specified card-side functions must have this before it:**
  HSTREAMS_EXPORT
- **Compile your sink-side app.**
  - Example: Use Intel's Composer xe compiler with the –mmic option
- **Use dynamic linking for host- and sink-side code**
  - Host: `-lcoi_host –lhstreams_source`
  - Device: `-fPIC –shared –rdynamic –lcoi_device -lhstreams_sink –Wl,-soname,<module_name>_mic.so`
- **When running**
  - Point SINK_LD_LIBRARY_PATH to MIC dependencies and to hstreams dynamic lib

API DETAILS

THE NEW CENTER OF POSSIBILITY

API details

- “app APIs” and “core APIs”
- Listing of APIs
- Layering
- API details

hStreams app APIs, for convenience

- **Wrapped and simplified core functions**
  - **hStreams_app_init** - initialize all domains; assumes homogeneous
  - hStreams_app_init_domains - initialize selected hetero domains
  - **hStreams_app_fini** - finalize
  - **hStreams_app_create_buf** - create buffer from host memory at all sinks
  - **hStreams_app_xfer_memory** – move data among buffers within a stream
  - **hStreams_app_invoke** – remote invocation of user-defined functions
  - hStreams_app_stream_sync - wait for all in stream to complete
  - hStreams_app_thread_sync - wait for all streams in thread to complete
  - hStreams_app_event_wait - wait on 1 or more events in source thread
  - hStreams_app_event_wait_in_stream - wait on 1 or more events in stream

hStreams app APIs, common building blocks

- **Common building blocks**
  - hStreams_app_memset
  - hStreams_app_memcpy
- **Intel® MKL CBLAS routines**
  - hStreams_app_sgemm
  - hStreams_app_dgemm
  - hStreams_app_cgemm
  - hStreams_app_zgemm

hStreams core APIs, part I

Those that are most likely to be used with app APIs are highlighted in red.

- **General**
  - hStreams_Init – initialization
  - hStreams_IsInitialized – check initialization
  - hStreams_Fini – finalization
- **Domains**
  - hStreams_GetNumPhysDomains – number of physical domains (cards)
  - hStreams_GetPhysDomainDetails – details per physical domain
  - hStreams_GetAvailable – CPU mask of available HW threads in a physical domain
  - hStreams_AddLogDomain – add a logical domain, if not with app_init APIs
  - hStreams_RmLogDomains – remove a list of logical domains
  - hStreams_GetNumLogDomains – number logical domains
  - hStreams_GetLogDomainIDList – list of logical domains
  - hStreams_GetLogDomainDetails – details per logical domain
- **Stream management**
  - hStreams_StreamCreate - register logical streams, provide card and CPU mask
  - hStreams_StreamDestroy - unregister logical streams
  - hStreams_GetNumLogStreams – number of logical streams
  - hStreams_GetLogStreamIDList – list of logical streams
  - hStreams_GetLogStreamDetails – details per logical stream

hStreams core APIs, part II

The functionality of these APIs is typically well covered by app APIs

- **Stream usage**
  - hStreams_EnqueueCompute - queue up compute work in a logical stream
  - hStreams_EnqueueData1D - queue up 1-dimensional data xfer work in a logical stream
- **Sync**
  - hStreams_StreamSynchronize - block until all actions in logical stream complete
  - hStreams_ThreadSynchronize - block until all actions in all logical streams complete
  - hStreams_EventWait - enforce a dependence in source thread, using one ore more events
  - hStreams_EventStreamWait - enforce a dependence in a stream, using one or more events and/or addresses to depend on
- **Memory Management**
  - hStreams_Alloc1D - allocate a 1-dimensional data buffer that may span all domains
  - hStreams_DeAlloc - deallocate and destroy a data buffer that may span all domains
- **Error Handling**
  - hStreams_GetLastError - report last COIERROR across all hStreams
  - hStreams_ClearLastError - clear last COIERROR across all hStreams

APIs deprecated since Intel® MPSS 3.4

We are seeking to improve the quality and intuitiveness of our product. To that end, we made some changes since the MPSS 3.5 hStreams rev.

As the product matures, we expect to place a high value on backwards compatibility.

See the hStreams_Porting_Guide.pdf for a description of changes since Intel® MPSS 3.5, including

- API changes to operand order
  - app_xfer: write address before read address
  - app_xfer, app_*gemm, app_mem*: logical stream ID first
  - app_invoke: completion event before return info
- New APIs
  - Alloc1DEx
  - EnqueueDataXDomain
- Otherwise modified
  - GetVersion returns value in a different format

Layering of hStreams APIs

- **Higher level**
  - Test apps – basic_perf, matMult, test_app, future Intel® MKL offering
  - Convenience functions, app_api
- **Dynamic partitioning**
  - Future extension, communicates with other processes
  - Then implements the partitioning with core implementation APIs
- **Core implementation: common, source and sink**
- **Internal files**

Common sequences: app API (1/3)

- **app_init or app_init_domains**
  - Uses all physical domains, with 1 logical domain per physical domain
    - *Card 0 will be physical domain 0 and logical domain 1*
    - *Card 1 will be physical domain 1 and logical domain 2*
    - *The number of cards can be limited to N by using*
      - *hStreams_GetCurrentOptions(&CurrentOptions, sizeof(HSTR_OPTIONS))*
      - *CurrentOptions.phys_domain_limit = N*
      - *hStreams_SetOptions(CurrentOptions)*
- **Specify the number of streams used, uniformly in every logical domain**
  - If you want this to be non-uniform, use app_init_domains instead
- **This already sets up logical domains, so no need to add them**
- **Specify the number of logical streams per stream, e.g. 1 or # tiles**
- **This already sets up logical streams, so no need to add them**
  - With app_init*, they are numbered starting at 0.

Macros and types

A common macro

HSTR_RESULT – wraps hStreams calls, and manages error reporting

Some of the commonly-used types are described below

HSTR_PHYS_DOM	physical domain; special value: HSTR_SRC_PHYS_DOM=-1
HSTR_LOG_DOM	logical domain
HSTR_LOG_STR	logical stream; physical streams are not visible
HSTR_ISA_TYPE	describes the target ISA
HSTR_XFER_DIRECTION	transfer direction between source and sink
HSTR_EVENT	completion event
HSTR_CPU_MASK	bit mask for CPUs; modeled after Linux CPU masks
HSTR_OVERLAP_TYPE	describes how a pair of CPU masks overlap

Mask utility functions

HSTR_CPU_<function>	where function is ISSET, SET, ZERO, AND, XOR, OR, COUNT EQUAL, XLATE, XLATE_EX – convenience functions for evaluating and manipulating CPU masks

Common sequences: app API (2/3)

- **Allocate data arrays on host with malloc**
    - It's best for performance to align these to a 64B boundary
- **app_create_buf**
    - Pass in the base address and size of the host-allocated memory
    - This will instantiate the buffer on all domains, which is a necessary prerequisite to data transfers or compute references. hStreams_Alloc1DEx can do selective instantiation.
- **app_xfer_memory**
    - Transfer a block from anywhere inside an allocated buffer to anywhere else inside any (same or different) allocated buffer
    - The direction of transfer refers to endpoints of the stream, i.e. source and the stream the logical stream is associated with
    - The last parameter is a completion event, so actions outside this stream can be made to wait on the transfer
    - Please don't confuse this with hStreams_app_memcpy, which does a sink-side call to the memcpy function

Common sequences: app API (3/3)

- **hStreams_app_invoke or hStreams_app_*gemm or hStreams_app_mem***
  - Invoke a function on the sink end of the logical stream
  - Pass arguments to a source-side thunk which does marshalling
  - Arguments are demarshalled by a sink-side thunk
  - Thunks are already provided for the *gemm and mem* functions, but you have to write your own for a user-provided function. See reference codes for examples in how to do this.
- **hStreams_app_fini**
  - Clean up resources

Common sequences: app + core API

- **For manual management of logical domains**
- **hStreams_app_init - initialization**
- **hStreams_GetNumPhysDomains – see how many**
- **hStreams_GetAvailable – get available mask**
- **For each domain you wish to add**
  - Partition the available mask manually – see ref codes
  - hStreams_AddLogDomain
  - For each stream within that logical domain
    - *hStreams_StreamCreate*
- **For each domain you subsequently wish to remove**
  - hStreams_RmLogDomain
    - *This removes all logical and physical streams in that domain*
- **Allocate memory, use streams**

If physical resources unknown

- **There may be some variability in the number of physical domains in your deployments**
  - You can check with **hStreams_GetNumPhysDomains**
  - If those are heterogeneous, use **hStreams_GetPhysDomainDetails** to get the properties of each physical domain
  - You can use that to guide parameter selection for **app_init_domains**
- **The assignment of logical domains to physical domains may be uneven**
  - Assignment is round robin, starting with physical domain 0
  - The mapping of logical domains may be checked using
    - *hStreams_GetNumLogDomains to get the number*
    - *hStreams_GetLogDomainIDList to get the individual IDs*
    - *hStreams_GetLogDomainDetails to get CPU mask and physical ID*
    - *hStreams_GetNumLogStreams to get number of streams in a domain*
    - *hStreams_GetLogStreamIDList to get the individual IDs*
    - *hStreams_GetLogStreamDetails to get CPU mask and logical domain ID*

app_*: Initialization and finalization

```
HSTR_RESULT hStreams_app_init(
        uint32_t            in_StreamsPerDomain,
        uint32_t            in_LogStreamOversubscription);
HSTR_RESULT hStreams_app_init_domains(
        uint32_t            in_NumLogDomains,
        uint32_t             *in_pStreamsPerDomain,
        uint32_t            in_LogStreamOversubscription);
HSTR_RESULT hStreams_app_fini();
```

- **Only need one init/fini pair once per host process, but can repeat**
  - Init before any other APIs, fini to flush IO (e.g. printfs)
- **Initialize per-process structures**
  - Based on querying device info, avoids OS/system threads
- **The _domains version allows skipping domains and variation in streams per domain that a hetero target system may need**
- **in_LogStreamOversubscription allows oversubscription of a stream without OS-level oversubscription**
- **The assignment of logical streams to physical streams is round-robin**

app_*: Buffer creation

```
HSTR_RESULT hStreams_app_create_buf(
        void              *in_BufAddr,
        const uint64_t    in_NumBytes);
```

- in_pBufAddr is a host-side, user-malloc'd buffer that the hStreams buffer gets created from
- Buffers are shared across all streams in the same domain
- Single data structures should not span multiple buffers
- Buffers define the granularity of dependence management
  - Making them finer can increase concurrency
  - Making them gratuitously fine may introduce unnecessary overhead
- Use hStreams_Alloc1DEx for selective instantiation or for manipulating buffer properties
  - Host instances of buffers created with default properties by hStreams_Alloc1DEx are not physically pinned

app_*: Data movement

```
HSTR_RESULT hStreams_app_xfer_memory(
        HSTR_LOG_STR          in_LogStreamID,
        void                 *in_pWriteAddr,
        void                 *in_pReadAddr,
        uint64_t              in_NumBytes,
        HSTR_XFER_DIRECTION   in_XferDirection,
        HSTR_EVENT           *out_pEvent);
```

- Source and destination objects must fit entirely within a buffer
- Source and destination addresses can each have arbitrary offset within their enclosing buffers, can be different from each other
- in_XferDirection specified the direction
- out_pEvent is an opaque completion event – see sync

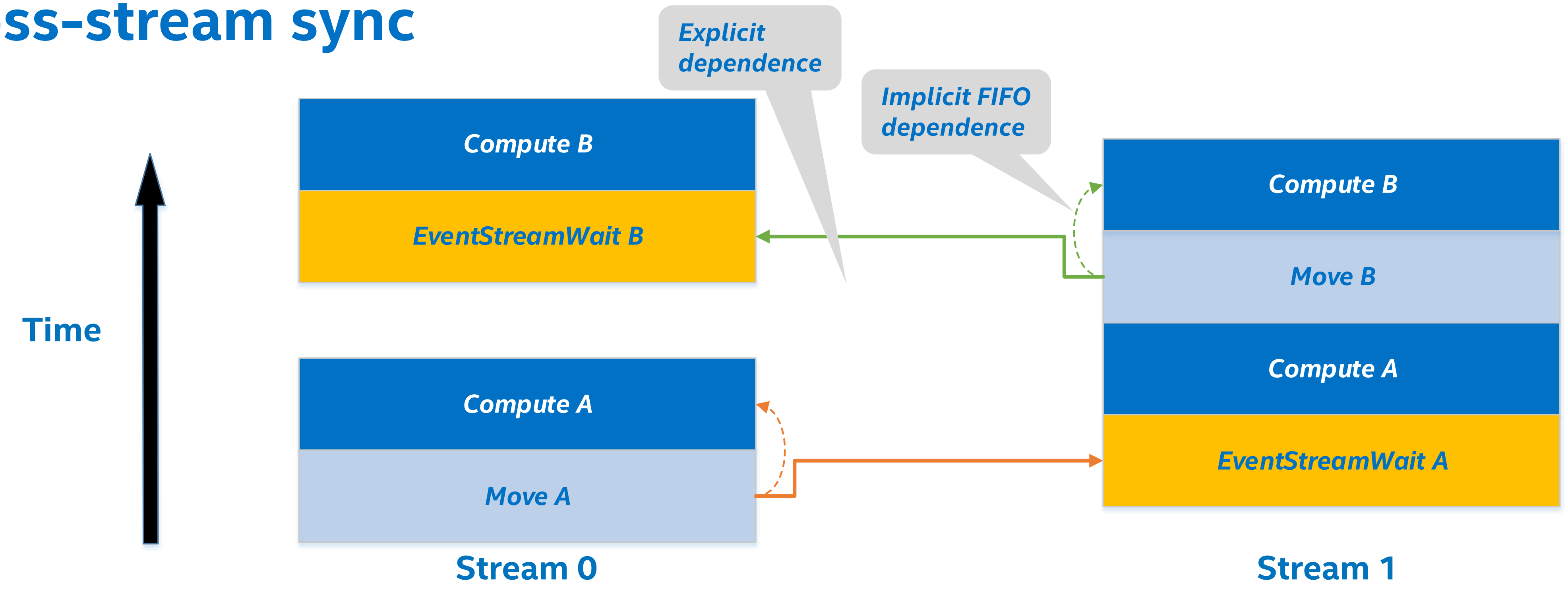
- Use hStreams_EnqueueDataXDomain1D for cross-domain transfers
- Note that the operand order changed from MPSS 3.5

app_*: Synchronization

```
HSTR_RESULT hStreams_app_stream_sync(
          HSTR_LOG_STR  in_LogStreamID);
HSTR_RESULT hStreams_app_thread_sync();
HSTR_RESULT hStreams_app_event_wait (
          uint32_t          in_NumEvents,
          HSTR_EVENT        *in_pEvents);
HSTR_RESULT hStreams_app_event_wait_in_stream (
          HSTR_LOG_STR  in_LogStreamID,
          uint32_t          in_NumEvents,
          HSTR_EVENT        *in_pEvents,
          uint32_t          in_NumAddresses,
          void             *in_pAddresses,
          HSTR_EVENT        *out_pEvent);
```

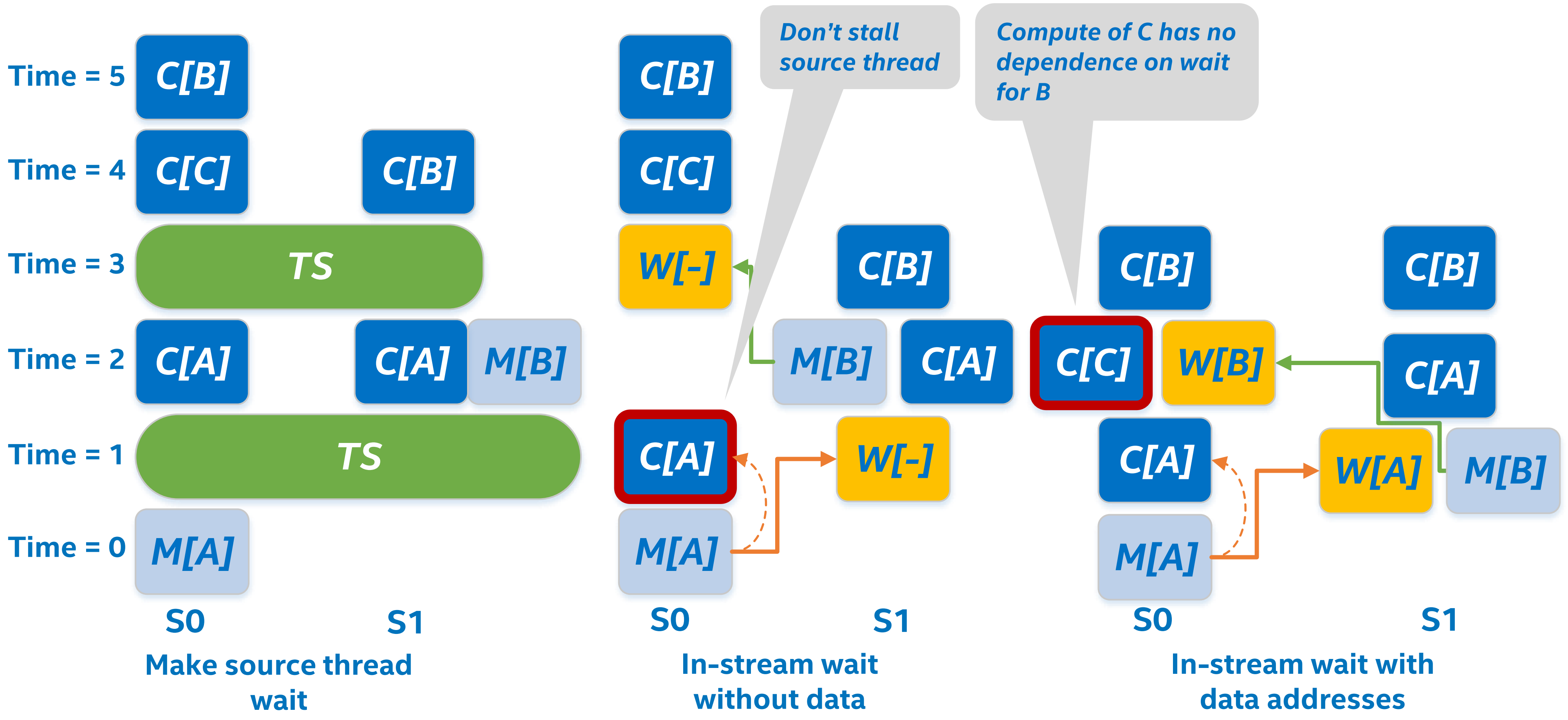
- Sync can happen within a stream, or across all streams in a source thread
- The event to be waited on is specified with an opaque handle
- Events may be waited upon in the source thread or a given stream

Cross-stream sync



- Streams 0 and 1 map to same domain, share data
- Explicit sync needed for cross-stream dependence of stream 1’s compute A on stream 0’s move A
- Move B in stream 1 would be control dependent on EventStreamWait A, unless addresses (for just B) are specified

Cross-stream sync scenarios



Key:
C – compute
M – move
S# – stream
TS – ThreadSync
W[x] – Wait for x

- In-stream waits don't block source thread so compute of A moves earlier in stream 0
- Specifying data for wait allows compute of C to be concurrent with wait in stream 0

app_*: General invocation

```
HSTR_RESULT hStreams_app_invoke(
        HSTR_LOG_STR          in_LogStreamID,
        const char           *in_pFuncName,
        uint32_t              in_NumScalarArgs,
        uint32_t              in_NumHeapArgs,
        uint64_t             *in_pArgs,
        HSTR_EVENT          **out_pEvent,
        void                 *out_pReturnValue,
        uint32_t              in_ReturnValueSize);
```

- The sink-side function is specified by name
- The number of scalar, then heap arguments that are packed into the array are given
- Pre-allocated space in the source for a return value may be given
- Use of a return value may incur noticeable extra costs
- The completion event can be used with sync APIs
- Note that the operand order changed from MPSS 3.5

app_*: Common functions 1/2

```
HSTR_RESULT hStreams_app_memset(
          HSTR_LOG_STR                  in_LogStreamID,
          void                         *in_pWriteAddr,
          int                           in_Val,
          uint64_t                      in_NumBytes,
          HSTR_EVENT                   *out_pEvent);
HSTR_RESULT hStreams_app_memcpy(
          HSTR_LOG_STR                  in_LogStreamID,
          void                         *in_pWriteAddr,
          void                         *in_pReadAddr,
          uint64_t                      in_NumBytes,
          HSTR_EVENT                   *out_pEvent);
```

- These are available in the pre-compiled hStreams binary
- The completion event can be used with sync APIs
- Please note that these arguments are in the same order as the standard memset and memcpy functions. The underlying _sink forms of these functions, not shown here, list scalar arguments before heap arguments.
- Note that the operand order changed from MPSS 3.5

app_*: Common functions 2/2

```
HSTR_RESULT hStreams_app_sgemm(
 const HSTR_LOG_STR LogStream,
  const CBLAS_ORDER Order,
  const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
  const MKL_INT M, const MKL_INT N, const MKL_INT K,
  const uint64_t alpha,
  const float *A, const MKL_INT lda,
  const float *B, const MKL_INT ldb, const uint64_t beta,
  float *C, const MKL_INT ldc,
 HSTR_EVENT* out_pEvent);
```

- Also dgemm (doubles for A,B,C)
- And cgemm, zgemm (const void* for A,B,C)
- These are available in the pre-compiled hStreams binary
- The completion event can be used with sync APIs
- Please note that these arguments are in the same order as the corresponding Intel® MKL functions. The underlying _sink forms of these functions, not shown here, list scalar arguments before heap arguments.

- Note that the operand order changed from MPSS 3.5

Core APIs: Initialization, finalization

- **HSTR_RESULT hStreams_Init()**
  - Once per host process, but there can be repeated Init/Fini calls
  - Before any other APIs
  - Initialize per-process structures
  - Queries device info
  - app_init and app_init_domains can be called after this, but this is not a pre-requisite
- **HSTR_RESULT hStreams_IsInitialized()**
  - Checks if already initialized
- **HSTR_RESULT hStreams_Fini()**
  - Once per host process, but there can be repeated Init/Fini calls
  - Destroy per-process structures if expected to no longer be used

Core APIs: Phys Domain Enumeration (1/2)

```
HSTR_RESULT hStreams_GetNumPhysDomains(
        uint32_t        * out_pNumDomains,
        uint32_t        * out_pActiveDomains,
        bool            * out_pHomogeneous)
```

- Must be after initialization, results won't change across invocations
- Call first to get number of visible physical domains.
- Homogeneous indicates whether all domains that are available for hStreams to use are have the same ISA, number of threads, etc.

```
HSTR_RESULT hStreams_GetPhysDomainDetails(
        HSTR_PHYS_DOM       in_PhysDomain,
        uint32_t            * out_pNumThreads,
        HSTR_ISA_TYPE       * out_pISA,
        uint32_t            * out_pCoreMaxMHz,
        HSTR_CPU_MASK       out_MaxCPUmask,
        HSTR_CPU_MASK       out_AvoidCPUmask,  // OS threads
        uint64_t            * out_pSupportedMemTypes,
        uint64_t            out_pPhysicalBytesPerMemType[
                               HSTR_MEM_TYPE_SIZE])
```

- 0-based numbering based on PCIe enumeration for cards, or other physical domains
- Provides info for a specific physical domain

Core APIs: Phys Domain Enumeration (2/2)

```
HSTR_RESULT hStreams_GetAvailable(
        HSTR_PHYS_DOM  in_PhysDomainID,
        HSTR_CPU_MASK  out_AvailableCPUmask);
```

- Returns cpu set that's currently not allocated from this process
- Must be after initialization
- Results change as streams are created and destroyed

Core APIs: Managing domains

- **Domains and streams can be managed automatically**
  - Created by hStreams_app_init*
  - Destroyed by hStreams_app_fini
- **Or manually**

```
HSTR_RESULT hStreams_AddLogDomain(
        HSTR_PHYS_DOM        in_PhysDomainID,
        HSTR_CPU_MASK        in_CPUmask,
        HSTR_LOG_DOM        *out_pLogDomainID,
        HSTR_OVERLAP_TYPE  *out_pOverlap);
```

  - Adds a single logical domain with a specified mask
  - Must be after initialization
  - Whether and how (no, exact, partial) it overlaps is returned

```
HSTR_RESULT hStreams_RmLogDomain(
        uint32_t                   in_NumLogDomains,
        HSTR_LOG_DOM              *in_pLogDomainIDs);
```

  - Remove in_NumLogDomains domain and their logical streams

Core APIs: Managing domains and streams

- **They can be incrementally removed and added back**
  - Initialize, remove
    - *hStreams_app_init(1,1)*
      - *On 2 cards creates logical domains 1, 2, streams 0,1,2,3*
  - HSTR_LOG_DOM LogDomIDs[2] = {1,2};
  - hStreams_RmLogDomains(2,LogDomIDs)
    - *Removes this logical streams too*
  - Then sample option A: explicit domain and stream creation
    - *hStreams_AddLogDomain(0,mask,&DomID,&Overlap), hStreams_StreamCreate(0,DomID,mask)*
  - Or sample option B: Partial use of app APIs
    - *uint32_t PlacesPerDomainArray[1] = {1};*
    - *hStreams_app_init_domains(1, PlacesPerDomainArray, 1)*
      - *If streams are not removed, e.g. by removing logical domains, before app_init_domains, trying to add the same logical domain index will produce an HSTR_RESULT_ALREADY_FOUND error*

Core APIs: Logical Domain Enumeration

```
HSTR_RESULT hStreams_GetNumLogDomains(
        HSTR_PHYS_DOM            in_PhysDomainID,
        uint32_t                *out_pNumLogDomains)
```

- Must be after initialization, results won't change across invocations
- Call before hStreams_GetLogDomainDetails to get number of logical domains.

```
HSTR_RESULT hStreams_GetLogDomainIDList(
        HSTR_PHYS_DOM            in_PhysDomainID,
        uint32_t                 in_NumLogDomains,
        HSTR_LOG_DOM            *out_pLogDomainIDs)
```

- Lists the 0-based logical domain IDs that were generated upon their addition

```
HSTR_RESULT hStreams_GetLogDomainDetails(
        HSTR_LOG_DOM             in_LogDomainID,
        HSTR_PHYS_DOM           *out_pPhysDomainID,
        HSTR_CPU_MASK            out_CPUmask)
```

- Provides info for a specific logical domain
- 0-based numbering based on PCIe enumeration for cards, or other physical domains, except that the source physical domain is HSTR_SRC_PHYS_DOMAIN (-1)

Core APIs: Logical Stream Enumeration

```
HSTR_RESULT hStreams_GetNumLogStreams(
        HSTR_LOG_DOM          in_LogDomainID,
        uint32_t              *out_pNumLogStreams)
```

- Must be after initialization, results won't change across invocations
- Call before hStreams_GetLogDomainDetails to get number of logical domains.

```
HSTR_RESULT hStreams_GetLogStreamIDList(
        HSTR_LOG_DOM          in_LogDomainID,
        uint32_t              in_NumLogStreams,
        HSTR_LOG_STR          *out_pLogStreamIDs)
```

- Lists the 0-based logical domain IDs that were generated upon their addition

```
HSTR_RESULT hStreams_GetLogStreamDetails(
        HSTR_LOG_STR          in_LogStreamID,
        HSTR_LOG_DOM          *out_pLogDomainID,
        HSTR_CPU_MASK         out_CPUmask)
```

- Provides info for a specific logical stream
- Logical stream ID values are provided by the user. They are 64-bit values, which are amenable for use with addresses.
- With app_init*, logical stream ID values are numbered starting at 0.
- Note: there's an erratum with the MPSS 3.6 release, wherein GetLogStreamDetails has the LogDomainID as a non-pointer input parameter. What's listed above is the corrected version.

Core APIs: Creating and destroying streams

```
HSTR_RESULT hStreams_StreamCreate(
        HSTR_LOG_STR            in_LogStreamID,
        HSTR_LOG_DOM            in_LogDomainID,
        const HSTR_CPU_MASK     in_CPUmask);
```

- Create the logical stream on the specified domain with the desired cpu set. Complete control.
- Create the corresponding physical stream if necessary
- in_LogStreamID is a logical stream number
- Physical streams are neither named nor referenced

```
HSTR_RESULT hStreams_StreamDestroy(
           HSTR_LOG_STR         in_LogStreamID);
```

- Destroy logical stream, last one out destroys underlying physical stream and COI resources
- Note: This is required to flush stdout from card

Core APIs: Checking stream binding

```
HSTR_RESULT hStreams_GetLogDomainDetails(
        HSTR_LOG_DOM          in_LogDomainID,
        HSTR_PHYS_DOM        *out_pPhysDomain,
        HSTR_CPU_MASK         out_CPUmask);
```

- Query the runtime to reveal the binding of a logical stream to underlying resources, e.g. which card and threads
- This is predictable and does not change between app_init() and app_fini, or hStreams_StreamCreate and hStreams_StreamDestroy.
  - The assignment of logical streams to physical streams is round-robin

Core APIs: Memory management

```
HSTR_RESULT hStreams_Alloc1D(
        void*       in_BaseAddress,  // host-side buf to create from
        uint64_t    in_size)         // size of buffer in bytes
```

- in_BaseAddress is a host-side, user-malloc'd buffer that the buffer gets created from
- Buffers are instantiated in all logical domains
- Buffer instances are shared across all streams within a given logical domain

```
HSTR_RESULT hStreams_DeAlloc (
        void*       in_Address)      // address anywhere in buffer
```

- in_Address is a host-side, user-malloc'd buffer that the buffer gets created from

Core APIs: Enhanced memory management

```
HSTR_RESULT hStreams_Alloc1DEx(
        void                *in_BaseAddress,     // host-side buf to create from
        uint64_t             in_size,            // size of buffer in bytes
        HSTR_BUFFER_PROPS   *in_pBufferProps,    // size of buffer in bytes
        int64_t              in_NumLogDomains,   // length of array of IDs
        HSTR_LOG_DOM        *in_pLogDomainIDs)   // array of log domains to instantiate
```

- in_BaseAddress is a host-side, user-malloc'd buffer that the buffer gets created from
- Buffer instantiations are *added* to the logical domains which are listed by pLogDomainIDs
  - Pre-existing instances are not removed or changed
- Buffers are shared across all streams in the same logical domain
  - In MPSS 3.5, there was an erratum: they were shared across all streams in the same card
- See next page for a description of buffer properties
  - Default properties are: not aliased, not pinned, not incremental, not affinitized, normal type, mem alloc preferred.
  - If aliased is set, wherein the user guarantees that no two instances may ever need to have distinct values for the same address, duplicate allocation and redundant transfers are suppressed.

Buffer properties

```
typedef struct HSTR_BUFFER_PROPS {
  HSTR_MEM_TYPE     mem_type;                              // Memory type
  HSTR_MEM_ALLOC_POLICY  mem_alloc_policy;                 // Memory allocation policy
  uint64_t          flags;                                 // Bitmask – see HSTR_BUFFER_PROP_FLAGS.
} HSTR_BUFFER_PROPS;

typedef enum HSTR_BUFFER_PROP_FLAGS {

   // Buffer instances should be aliased when their logical domains are mapped to the same physical domain
   HSTR_BUF_PROP_ALIASED = 1,

   // The instance associated with HSTR_SRC_LOG_DOMAIN is pinned when
   //  buffer is created. Otherwise defer pinning until on-access demand.
   HSTR_BUF_PROP_SRC_PINNED = 2,

   // When a new logical domain is added, an instantiation of this buffer is automatically added for that log domain
   HSTR_BUF_PROP_INCREMENTAL = 4,

   // The first touch of each instantiation of this buffer is constrained to be performed by a thread that belongs
   //  to the CPU set of its logical domain.  Functionality of this flag is not implemented yet, Alloc1DEx
   //  returns HSTR_RESULT_NOT_IMPLEMENTED if that flag is set.
   HSTR_BUF_PROP_AFFINITIZED = 8,

} HSTR_BUFFER_PROP_FLAGS;
```

Core APIs: Data movement

```
HSTR_RESULT hStreams_EnqueueData1D(
    HSTR_LOG_STR   in_LogStreamID,
    void*          in_pWriteAddr,      // currently must be NULL
    void*          in_pReadAddr,       // where to copy from
    uint64_t       in_size,            // bytes to move
    HSTR_XFER_DIRECTION in_XferDirection, // dir of mov't
    HSTR_EVENT     *out_pEvent); // for async waits
```

- in_pReadAddr and in_pWriteADdr are host-proxy addresses, even if copying from sink
- Valid values for in_XferDirection are HSTR_SRC_TO_SINK and HSTR_SINK_TO_SRC

Core APIs: Data movement – more general

```
HSTR_RESULT hStreams_EnqueueDataXDomain1D(
     HSTR_LOG_STR                      in_LogStreamID,
     void                             *in_pWriteAddr,       // where to copy to
     void                             *in_pReadAddr,        // where to copy from
     uint64_t                          in_size,             // bytes to move
    HSTR_LOG_DOM                       in_destLogDomain,    // logical domain ID to move to
    HSTR_LOG_DOM                       in_srcLogDomain,     // logical domain ID to move from
    HSTR_EVENT                        *out_pEvent);         // for async waits
```

- This is a more-general form of data movement that EnqueueData1D, since *at most one* of the source and destination logical domains may be different than the logical domain of the source and sink ends of the stream.
- Use this form of the API to enable transfers among sinks, e.g. card-card transfers
- Cross-card transfers are subject to further improvements in performance

Core APIs: Invocation

```
HSTR_RESULT hStreams_EnqueueCompute(
        HSTR_LOG_STR        in_LogStreamID,
        const char         *in_pFunctionName,
        uint32_t            in_numScalarArgs,  // copied by value
        uint32_t            in_numHeapArgs,    // passed in bufs
        uint64_t            ** in_pArgs,       // user-created array
        HSTR_EVENT         *out_pEvent,        // for async waits
        void               *out_ReturnValue,   // for ret values
        size_t              in_ReturnValueSize) // size in bytes
```

- Name must match function in code compiled with -mmic
- Number of scalar and heap args must be specified
- Scalar and heap args packed into 64b array elements of in_pArgs: scalar first, then heap.
- Scalars are values, copied by value
- Heap args must be host (proxy) pointers

Core APIs: Synchronization

```
HSTR_RESULT  hStreams_ThreadSynchronize();
HSTR_RESULT  hStreams_StreamSynchronize(
             HSTR_LOG_STR  in_LogStreamID);
```

- Synchronize across all threads
- This syncs within each phsyical stream in implementation

Core APIs: Synchronization

```
HSTR_RESULT hStreams_EventWait(
        uint32_t        in_NumEvents,
        HSTR_EVENT      *in_pEvents,
        bool            in_WaitForAll,
        int32_t         in_TimeOutMilliSeconds,
        uint32_t        *out_pNumSignaled,
        uint32_t        *out_pSignaledIndices);
```

- Source waits on the completion of one or more events
- Can specify whether to wait for all, just at least one
- Time out of -1 is wait forever, 0 is polling, else something >1
- SignaledIndices is a packed array of events that completed

Core APIs: Synchronization

```
HSTR_RESULT hStreams_EventStreamWait(
        HSTR_LOG_STR   in_LogStreamID,
        uint32_t       in_NumEvents,
        HSTR_EVENT     *in_pEvents,
        uint32_t       in_NumAddresses,
        void           *in_pAddresses);
```

- **A given stream, not the entire source thread, waits on the completion of one or more events**
- **Presumed to wait for all, indefinitely**
- **The array of addresses indicate data dependences to resolve**
  - If NumAddresses == NULL, then this acts as a control dependence
  - If non-NULL, then only computes and data transfers that overlap with the buffers which enclose the listed addresses need to wait

BACKUP

THE NEW CENTER OF POSSIBILITY

Backup

- Terminology
- Notices and disclaimers

Terminology

- **Definitions of terms used throughout this deck**
  - **Physical [memory coherence] Domain –** set of resources across which memory is [efficiently] shared, e.g. a card or node in a cluster
  - **Logical Domain –** abstraction that users refer to; one or more logical domains can be bound to a physical domain
  - **Heterogeneous** – non-homogeneous with respect to number or capabilities of resources
  - **Oversubscribed** – multiple tasks mapped to the same resources, such that the OS gets involved in switching between them
  - **Partition** – division of compute resources into disjoint subsets
  - **Place** – subset of resources within a given domain (OpenMP4)
  - **Sink** – remote domain invoked from the source
  - **Source** – domain from which commands are issued
  - **Stream** – FIFO abstraction which is assigned to a stream and within which dependences are implicit. Only logical streams are exposed.
  - **Buffer** – allocated block of memory; all data transfers and computes operate onparts of these

Notices and disclaimers (1)

Notices and disclaimers (2)