Support for multi-operation stream architecture

Revision 0.1

Contents

[1. Generic Streams 1](#_Toc125051383)

[2. Design Goal 1](#_Toc125051384)

[3. Architecture Requirements 2](#_Toc125051385)

[4. Current Bridge stream Design 2](#_Toc125051386)

[5. Generic Stream HLD 7](#_Toc125051387)

[6. Force Usage of default stream: 9](#_Toc125051388)

[7. Different Modes of operation support - TBD 9](#_Toc125051389)

[References 10](#_Toc125051390)

1. Generic Streams

A stream is a queue of device work. host places the work in the queue and continues on immediately. device will schedule the work in the queue when the resources are free. Operations like kernel/copy/collective are placed within the stream. operation is in FIFO within stream and no need of explicit synchronization. operations in different streams are unordered and need explicit synchronization

A generic stream is an entity where all operation can be pushed to a stream irrespective of the type of operation. (Copy, kernel, or collective operation).

1. Design Goal
2. Support multi-functional streams:

To be more compliant with current industry user usage of streams, where a user can push any type of operation into a stream created. (Similar to CUDA). This will support an asynchronous stream API that allows the user to create a stream to perform any stream operation with the handle, meaning “memcpy”, “launch” and collective operations.

1. Allow user to create any operation and push to a stream. (Currently bridge supports only for compute stream) to enable parallel execution of operation that are not dependent. Example: we can run Copy\_H2D with non\_blocking =true on a user stream and manage his using stream/event apis and parallel run the next compute or copy operations.

Without Generic Stream:



With Generic Stream (using d2h copy with user stream) can be pipelined better



1. Architecture Requirements
   1. Provide torch.cuda.Stream and torch.cuda.Events equivalent on Habana

Provide torch.hpu.Stream and torch.hpu.Event, that can be used by python scripts.

* 1. Provide a python scope-based stream usage

PyTorch provides cuda streams using “with” statements. Provide same interface for Habana streams

* 1. Allow creation of multiple user streams

User should be able to create multiple streams, and not limited by the number of streams available on a Gaudi device. This necessitates multiplexing of user created streams over the available synapse streams

* 1. Same user stream to gather compute and network ops
* DeepSpeed uses the same user streams to invoke both compute and network ops. Same should be provided for Habana, and internally the ops should be sent to the compute or network synapse streams, as appropriate. Note: This isn’t required to be handled by the bridge. The network ops are separated out to collective stream by the framework via distributed process group implementation.
  1. Retain execution order:
* The execution order across multiple streams, and compute and network ops must be retained as provided from the script execution.

1. Current Bridge stream Design

* Different dedicated Habana streams are created in device is acquired and object is created. one dedicated stream for copy\_D2D, copy\_D2H, copy\_H2D and compute. (This is the default stream).
* Bridge takes the responsibility to synchronize between the different stream’s operations using the record Events and waitForStreamEvent synapse APIs.
* User stream are only support for compute stream and users can create unlimited stream. During the first call to create a user stream, multiple streams are created for compute stream based on the stream limitation by HW type. The bridge will do a round-robin based on the actual compute stream supported by the HW.
* Each stream has a dedicated gc thread to check the stream status and to handle events associated to the stream.
* Creates a thread local current\_stream variable (since for Habana, each card has a separate process – so every process has a single device)
* Provide a StreamID, that encapsulates the type (default or non-default) and the ID (default ID is always 0)
* Stream ID assignment:
  + currently we are using the similar approach as in CUDA.
  + How stream ID are assigned:
    - -- 57 bits -- -- 5 bits ----- -- 3 bits --

zeros stream id index StreamIdType

Where StreamIdType:

1. default stream or externally allocated if id[63:3] != 0
2. 001 = low priority stream
3. 010 = high priority stream /\*not used \*/

This stream id is mapped to the compute 0 and compute 1(for Gaudi1) and user stream will be gets this in round robin fashion based on the actual limit.

* Interface:
  + HPUStream class: Value object representing a HPU stream. This is just a wrapper around c10::Stream, but it comes with a little extra HPU-specific

functionality (conversion to device stream), and a guarantee that the wrapped c10::Stream really is a HPU stream.

* + Synchronize () – synchronize a stream.
  + Query () – Query the status of the stream
  + Torch API
    - TORCH\_API HPUStream getStreamFromPool (const bool isHighPriority = false, DeviceIndex device = -1): Get a new stream from the HPU stream pool. streams are preallocated from the pool and returned in a round-robin fashion. For HPU device index will be 0 always as every node has separate copy of synapse lib and will get device with index 0. Priority is not supported and unused now.
    - TORCH\_API HPUStream getDefaultHPUStream (DeviceIndex device\_index = -1): Get the default HPU stream, for the HPU device, or for the current device if no device index is passed. The default stream is where most computation occurs when there is no explicitly using streams.
    - TORCH\_API HPUStream getCurrentHPUStream (DeviceIndex device\_index = -1): Get the current HPU stream, for the HPU device, or for the current device if no device index is passed. The current HPU stream will usually be the default HPU stream for the device, but it may be different if someone called 'setCurrentHPUStream' or used 'StreamGuard' or 'HPUStreamGuard'.
    - TORCH\_API void setCurrentHPUStream (HPUStream stream): Set the current stream on the device of the passed in stream to be the passed in stream. This function has \*nothing\* to do with the current device: it toggles the current stream of the device of the passed stream.
  + Python Interface API
    - class Stream(object): Wrapper around a HPU stream. A HPU stream is a linear sequence of execution that belongs to a specific device, independent from other streams.
      * query(self): Checks if all the work submitted on the stream has been completed. returns a Boolean indicating if all kernels in this stream are completed.
      * synchronize(self): Wait for all the kernels in this stream to complete.
      * record\_event(self, event=None): Records an event and returns an event.
      * wait\_event(self, event): Makes all future work submitted to the stream wait for an event. This function returns without waiting for :attr:`event`: only future operations are affected.
      * wait\_stream(self, stream): Synchronizes with another stream. All future work submitted to this stream will wait until all kernels submitted to a given stream at the time of call complete.
    - class StreamContext(object): Context-manager that selects a given stream. All hpu kernels queued within its context will be enqueued on a selected stream. (With:)
    - stream(stream) -> StreamContext: Wrapper around the Context-manager StreamContext that selects a given stream.
    - set\_stream(in\_stream): Sets the current stream. This is a wrapper API to set the stream.
    - current\_stream(): Gets the current stream.
    - default\_stream(): Gets the default stream on HPU device. This is a wrapper API to get the stream.
    - get\_stream\_info(stream:Stream): Gets the info for HPU stream
    - class Event (): Wrapper around a HPU event. events are synchronization markers that can be used to monitor the device's progress, to accurately measure timing, and to synchronize HPU streams. After creation, only streams on the same device may record the event. This takes enable\_timing (bool, optional) arguments to indicates if the event should measure time
      * + record (self, stream=None): Records the event in a given stream. Uses ``htorch.hpu.current\_stream()`` if no stream is specified.
        + wait(self, stream=None): Makes all future work submitted to the given stream wait for this event. Use ``htorch.hpu.current\_stream()`` if no stream is specified.
        + query(self): Checks if all work currently captured by event has completed. A Boolean indicating if all work currently captured by event has completed.
        + elapsed\_time(self, other): Returns the time elapsed in milliseconds after the event was recorded and before the end\_event was recorded.
        + synchronize(self): Waits for the event to complete. Waits until the completion of all work currently captured in this event. This prevents the CPU thread from proceeding until the event completes.
* Stream Creation in Bridge:
  + Initial stream creation in bridge



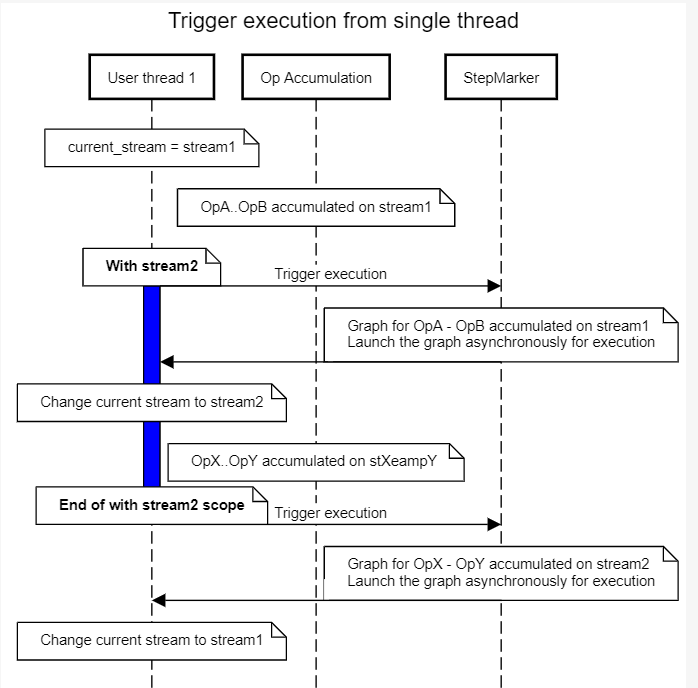
When device object is created, all the basic streams are created one for each operation type and used when no streams are specified.

* + User stream Creation:



All the compute device stream are created when user calls first time the Stream creation for the first time and these streams are stored in device with the hpu stream index. When next time the new stream is requested from user – based on the availability the actual HW streams are assigned in round robin fashion.

* Single Thread of Execution with Multiple HPU streams:

  
When the streams are used from same thread, execution can be triggered every time a stream is switched. Anytime current\_stream is modified (possibly from a StreamContext \_\_entry\_\_ function), trigger a mark\_step on the previous current\_stream. At the end of current\_stream scope with \_\_exit\_\_, mark\_step can be triggered for the new stream before restoring the previous current\_stream.

* Handling Multiple Stream of execution:
  + Once graphs are identified per stream, they need to be launched on the intended streams
  + The required stream ID for the execution will be sent to lowering – which launches synapse recipe
  + For all input tensors, perform the usual wait with synStreamWaitEvent, for the compute stream where the launch would take place.
  + If the tensor is being produced from any other stream, an event would be recorded on that stream as it is done today
  + Given the tensor address, this event would be found and the synStreamWaitEvent would make the launch stream to wait on the event
  + No change required in the current event handling for input data dependency, except for using the target compute stream
  + The recipe launch (synLaunchWithExternalEventsBase) must be done for the given synapse stream
  + Once the launch is done, add synapse event with synEventRecord
    - All the output tensor addresses will be mapped to the event
    - If there is a timer event associated to this Launch, it should use the timer event.
    - If there user event for this Launch, it should use the user event, so that the event query will reflect the correct status.
* Forced mark \_step
  + Handling multiple streams in lazy mode would require forced mark step from the bridge at several points for ensuring the semantics are preserved. In CUDA eager execution, all ops are known to have been launched on the device, whereas in lazy, this isn’t true. The ops are accumulated till an execution is triggered. The current execution trigger points in the bridge are
    - User driven mark\_step()
    - Data transfer from device to CPU (D2H) – triggered by .to(‘cpu’) or other framework code that requires the data to be fetched to host. These could also be triggered due to data dependent control flow or asserts in the script
    - Non-inferable op accumulation – requires execution to get the true output tensor shape and proceed forward
  + For supporting multiple streams, further mark\_step points would be
    - Each time a stream context changes from a user thread of execution
    - For all wait/sync/record stream/events triggered from the user script
* Support multiple streams from multiple threads:
  + Not supported now.
* Limitations:
  + Backward ops will always be on default stream. (Not supported)
  + Priority not supported.
  + Only Compute stream can be created by user.
  + D2D, for lazy mode this will be part of the compute using memcopy node. This cannot be initiated on user stream as it will break the graph. So, it will be part of the compute.

1. Generic Stream HLD

* Default Stream:
  + During device object initialization, only one stream which a default stream will be created. (This is a Multi operation stream)
  + All the operation can be submitted through this stream user has not specified any stream.
  + For default stream, all the operation dependency will be handled by synapse.
* User streams:
  + Users stream are created when request and this will directly call synapse API. synapse will manage the default stream affinity. stream affinity is not used.
  + Like compute, for copy/collective need to get the current stream and use that stream handle for copy/collective operation.
* Both User Stream & Default stream, will use the synapse Helper stream/event/SEM to create/manage stream.
* Users stream are created when requested via stream API and this will directly call synapse synStreamCreateGeneric().
* Collective Streams:
  + Collective streams will use create a UserStream as like Cuda/Gloo using getStreamFromPool ().
  + CUDA, uses high priority stream for this purpose. for HPU, we will use the normal stream as priority is not supported.
  + TDB - to look at the affinity, if it can be used to dedicate a stream for Network
* Stream ID assignment:
  + stream id is uint64 and that many streams index can be generated.
  + streamID to StreamHandle map will be maintained in device class.
  + Stream Index 000 - is a default stream.
* Interface API changes:
  + There is no change with respect python or Torch API for generic stream.
* Implicit/Explicit Synchronization:
  + Already this is handled in the Bridge. no change with respect to generic stream.
* Scalar copies:
  + Will use the default stream.
  + TBD to check if we can use current stream.
* Default Stream Creation:



Only one Default Generic stream will be created when device object is initialized. If the stream is not specified by the user, default stream will be used for all the operations.

* User Stream Creation:



User stream are created using the new generic stream synStreamCreateGeneric () to create a new stream when a user thread request for a new stream.

* Copy Stream handling with user stream:
  + Get the Current HPU stream for the copy operation. always perform the copy using the current stream.
  + Pass this to the device copy to use this specific stream handle.
  + Handling of user specific events:

TBD , need to check how this user\_event need to be used for copy streams

* Limitation:
  + Backward ops will always be on default stream.
  + Priority not supported.
  + D2D, for lazy mode this will be part of the compute using memcopy node. This cannot be initiated on user stream as it will break the graph. So, it will be part of the compute.

1. Force Usage of default stream:

When PT\_HPU\_FORCE\_DEFAULT\_STREAM is set, all operations are performed on the default stream. This is provided for debug purpose.

1. Different Modes of operation support:
   * Lazy Mode: for stream synchronize/query () no change is expected, only copy/collective change are expected as explained above.
   * Eager Mode: need to pass the stream/ event information for synchronize/query/wait/record event. (This is required only for user events)
   * Pytorch 2.0: (TBD)
     + removal of Mark Step, but need to add function to flush the current launch queue and call synchronize/ wait event etc.
     + For user created events, this need to be passed to actual lunch execution thread and copy streams.
2. Opens:
   * + Handling of user created events and using them explicitly as a timer event or for explicit synchronization.

# References

1. [PyTorchHabanaUserStreamArchDraft.docx](https://habanalabs-my.sharepoint.com/:w:/g/personal/ssaraswati_habana_ai/EX9JLf57rxNLvDL2AudjyF4BUoloqxT3mTXA9PVe4_-nJA?xsdata=MDV8MDF8fGM4NWIwMTExMDc1NDQ0N2VmMmE5MDhkYTNkOGU2NTk1fDBkNGQ0NTM5MjEzYzRlZDhhMjUxZGM5NzY2YmExMjdhfDB8MHw2Mzc4ODk5NzkwMjUzMjI2NjV8R29vZHxWR1ZoYlhOVFpXTjFjbWwwZVZObGNuWnBZMlY4ZXlKV0lqb2lNQzR3TGpBd01EQWlMQ0pRSWpvaVYybHVNeklpTENKQlRpSTZJazkwYUdWeUlpd2lWMVFpT2pFeGZRPT18MXxNVGs2WXpoalptSTRZMlEyTURnMk5HRTJaV0ZrTWpneE5EZzNOMk5rTW1RME56bEFkR2h5WldGa0xuWXl8fA%3D%3D&sdata=cHFJUnRYNi9BNHBqMnFRQVNSd1FwSHptYnhqTnFQMUNXZ1EyOTNmTW5DYz0%3D&ovuser=0d4d4539-213c-4ed8-a251-dc9766ba127a%2Cdvarshney%40habana.ai)
2. <https://github.com/pytorch/pytorch/blob/88fca3be5924dd089235c72e651f3709e18f76b8/torch/cuda/__init__.py#L378>
3. <https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html>
4. <https://pytorch.org/docs/stable/notes/cuda.html#stream-semantics-of-backward-passes>
5. <https://pytorch.org/docs/stable/generated/torch.cuda.Stream.html>