HiHAT asynchronous operations

Wojtek Wasko, NVIDIA

March 20th, 2018
ONLINE

Background
Requirements and assumptions
Proposal
Questions?
Background

- Architecture vs. implementation
- Actions - invocation, data mgt, data movement, sync
- Action handles
  - Enables reference to and querying overall status of an action
  - Most common dependence link from one action to another
  - Some actions perform logical operations on ActionHndls, link multiple predecessors
- Sync objects
  - May be one or more per action (could be a graph), but one post-dominates all others
Requirements Overview

Action handles

A.1/ **Link actions** according to dependences via ActionHndl
   [ already covered by input dep in hhuInvoke/hhuCopy/… ]

A.2/ **Create logical AND/OR of multiple action handles**
   [ already covered by hhuSync(Any|All) ]

A.3/ **Query action status** via action handle

A.4/ Obtain and operate on an action’s underlying sync object

Interoperability on sync objects without HiHAT’s involvement

S.1/ Based on sync object’s memory semantics description

S.2/ Based on sync object’s type
Design

- What client (above HiHAT) owns
  - Pointer to ActionHndl

- What HiHAT owns
  - ActionHndl - created when submitted, destroyed with a Clean operation or explicitly
    - Members of ActionHndl, which may include sync object(s) should get cleaned up by destroy

- What implementations plugged in under HiHAT own
  - Sync object(s), which may be a member in ActionHndl
  - Management of dependences, evaluation of readiness to execute
    - Polling and triggering of sync objects; whether native or programmatic interfaces are used
Anatomy and ownership of an hhActionHndl

Illustrated by the case of a cudaEvent_t

- owned by the client
- owned by HiHAT
- owned by the implementation
PROPOSAL
A.1/ Link actions according to dependences via ActionHndl

Already there in the API. To list a few examples:

```c
hhRet hhuCopy(
    hhDataView dst,
    size_t dst_offset,
    hhDataView src,
    size_t src_offset,
    size_t num_bytes,
    hhExecPol exec_pol,
    hhExecCfg exec_cfg,
    hhResrcHndlSet exec_resrc,
    hhActionHndl input_dep,
    hhActionHndl *out_action_hndl);
```

```c
hhRet hhuInvoke(
    hhTaskHndl reg_task,
    void *dblob,
    hhExecPol exec_pol,
    hhExecCfg exec_cfg,
    hhResrcHndlSet exec_resrc,
    hhActionHndl input_dep,
    hhActionHndl *out_action_hndl);
```

```c
hhRet hhuUnregMem(
    hhDataView mem_hndl,
    hhExecPol exec_pol,
    hhExecCfg exec_cfg,
    hhResrcHndlSet exec_resrc,
    hhActionHndl input_dep,
    hhActionHndl *out_action_hndl);
```
A.2/ Create logical AND/OR of multiple action handles

hhRet hhuSyncAll(
    hhActionHndlSet input_deps,
    hhExecPol exec_pol,
    hhExecCfg exec_cfg,
    hhResrcHndlSet exec_resrc,
    hhActionHndl *out_action_hndl);

A logical AND of input_deps.

hhRet hhuSyncAny(
    hhActionHndlSet input_deps,
    hhExecPol exec_pol,
    hhExecCfg exec_cfg,
    hhResrcHndlSet exec_resrc,
    hhActionHndlSet *out_triggered,
    hhActionHndl *out_action_hndl);

A logical OR of input_deps.
A.3/ Query action status via action handle

// Retrieve status of an individual action handle
// New addition
hhRet  hhnActionHndlGetActionState(
    hhActionHndl hndl,
    hhActionStateEnum *out_state);

// Actively blocks until all actions are completed
// Already present.
hhRet  hhnSyncAll(
    hhActionHndlSet actions);

// Actively blocks until any of the actions is completed
// Returns the actions which have been triggered in out_triggered
// Already present.
hhRet  hhnSyncAny(
    hhActionHndlSet actions,
    hhActionHndlSet *out_triggered);
A.4/ Obtain and operate on an action’s underlying sync object

// Caller does not gain ownership of out_sync_obj_descr
// Expected users: client, implementation
// May fail if block_until_available == False and impl has not yet set the sync obj on an action hndl

hhRet hhneActionHndlGetPostdominatingSyncObj(
    hhActionHndl action_hndl,                  // input action handle
    bool block_until_available,               // whether to block until the sync obj is available
    hhSyncObjHndl *out_sync_obj,              // output sync object handle
    const hhSyncObjDescr **out_sync_obj_descr // output sync object description
);

// For the implementation to set the postdominating sync object on an action handle
//
// Caller does not cede ownership of sync_obj_descr
// Expected users: implementation

hhRet hhneActionHndlSetPostdominatingSyncObj(
    hhActionHndl action_hndl,                  // input action handle
    hhSyncObjHndl sync_obj,                    // input sync object
    const hhSyncObjDescr *sync_obj_descr       // input sync object description
);
12

**S.1/ Interop on sync objects based on sync object’s description**

```c
typedef struct hhSyncObjDescr {
    int64_t type_index;
    size_t sync_object_bytes;
    hhSyncTriggerKindEnum kind;
    size_t state_field_offset;
    size_t state_field_bytes;
    int64_t triggered_val;
    int64_t untriggered_val;
    int64_t uninit_val;

    hhRet (*query_syncobj_state_fn)(hhSyncObjHndl, hhSyncObjDescr *, hhSyncObjStateEnum*);
    hhRet (*free_syncobj_fn)(hhSyncObjHndl, hhSyncObjDescr *);
} hhSyncObjDescr;

typedef void* hhSyncObjHndl;

typedef enum {
    HH_TRIGGER_NOT_DESCRIBED,
    HH_TRIGGER_EQ,
    HH_TRIGGER_GE,
    HH_TRIGGER_GT,
    HH_TRIGGER_LE,
    HH_TRIGGER_LT,
    // G = greater, L = less
    // T = than, and E/EQ = equal
} hhSyncTriggerKindEnum;

typedef enum {
    HH_SYNCOBJ_UNINIT,
    HH_SYNCOBJ_UNTRIGGERED,
    HH_SYNCOBJ_TRIGGERED
} hhSyncObjStateEnum;
```
S.2/ Interop on sync objects based on sync object’s type

Allow HiHAT’s clients and implementations to cooperate “natively” on a synchronization object without HiHAT’s involvement, based on the clients’ knowledge of the nature of the synchronization object.

E.g.: Is the synchronization object a `cudaEvent_t`? If so, I want to use it directly (because e.g. my hardware understands it natively).

Facilitated by `int64_t hhSyncObjDescr::type_index`. 
type_index (1/2)

HiHAT needs a type index. How clients/implementations come up with that index is up to them.

As one of possible solutions, HiHAT will ship a header file that enumerates some possible synchronization object types and their descriptions. Same header will declare pointers to sync obj descriptors. <see next slide for a strawman>

HiHAT will not use this file internally.

The type is not registered with HiHAT per se.
hhSyncObjTypes.h:

```c
#ifndef HHH_SYNCOBJ_TYPES_
#define HHH_SYNCOBJ_TYPES_

enum hhSyncPrimitives {
    HHH_CUDA_EVENT_T,
    /*!< hhSyncObjHndl can be cast to cudaEvent_t */
    HHH_EGLSyncKHR,
    /*!< hhSyncObjHndl can be cast to EGLSyncKHR */
};

// fwd decl
struct hhSyncObjDescr;

extern const struct hhSyncObjDescr *HHH_CUDA_EVENT_T_DESCR;
extern const struct hhSyncObjDescr *HHH_EGLSyncKHR;
extern const struct hhSyncObjDescr *HHH_FILE_DESCRIPTOR_DESCR;

#endif // ndef HHH_SYNCOBJ_TYPES_
```

hhSyncObjTypes.cpp:

```c
#include "hhSyncObjTypes.h"
#include "<HiHAT header that defines struct hhSyncObjDescr>"
#include <cuda.h>

static hhRet __QueryCudaEvent(hhSyncObjHndl hndl, hhSyncObjDescr *descr, hhSyncObjStateEnum *out_state) {
    assert(descr->type_index == HHH_CUDA_EVENT_T);
    if (cudaSuccess == cudaEventQuery((cudaEvent_t)hndl)) {
        *out_state = HHH_SYNCOBJ_TRIGGERED;
    } else {
        *out_state = HHH_SYNCOBJ_UNTRIGGERED;
    }
    return HHRET_SUCCESS;
}

static hhRet __FreeCudaEvent(hhSyncObjHndl hndl, hhSyncObjDescr *descr) {
    assert(descr->type_index == HHH_CUDA_EVENT_T);
    cudaEventDestroy((cudaEvent_t)hndl);
    return HHRET_SUCCESS;
}

static const struct hhSyncObjDescr HHH_CUDA_EVENT_T_DESCR_STR =
    { HHH_CUDA_EVENT_T, sizeof(cudaEvent_t),
      HHH_TRIGGER_NOT_DESCR, 0, 0, 0, 0,
      __QueryCudaEvent, __FreeCudaEvent };

const struct hhSyncObjDescr *HHH_CUDA_EVENT_T_DESCR =
    &HHH_CUDA_EVENT_T_DESCR_STR;
```
SAMPLES
Sample 1/5: using \texttt{hhnActionHndlGetActionState()} [client code]

```c
int main() {
    // ...

    hhActionHndl action_hndl;
    CHECK_HIHAT(hhuInvoke( <...>, &action_hndl));

    hhActionStateEnum action_state;
    do {
        CHECK_HIHAT(hhnActionHndlGetActionState(action_hndl, &action_state));
    } while (action_state == HH_ACTION_PENDING);

    // ...
}
```
Sample 2/5: cudaEvent_t [client code]

```c
int main() {
    // ...

    hhActionHndl action_hndl;
    CHECK_HIHAT(hhuInvoke( <...>, &action_hndl));

    hhSyncObjHndl sync_obj;
    hhSyncObjDescr *sync_descr;
    CHECK_HIHAT(hhneActionHndlGetPostdominatingSyncObj(action_hndl, 1, &sync_obj, &sync_descr));

    if (sync_descr->type_index == HHH_CUDA_EVENT_T) {
        CHECK_CUDA(cudaEventSynchronize((cudaEvent_t) sync_obj));
    } else { /* default to querying HiHAT’s API */
        hhActionStateEnum action_state;
        do {
            CHECK_HIHAT(hhneActionHndlGetActionState(action_hndl, &action_state));
        } while (action_state != HH_ACTION_COMPLETED);
    }

    // ...
}
```
Sample 3/5: `cudaEvent_t` [implementation]

```c
hhInvokeCUDAImpl(..., hhActionHndl input_dep, ...) {
    hhSyncObjHndl sync_obj;
    hhSyncObjDescr *sync_descr;
    CHECK_HIHAT(hhneActionHndlGetPostdominatingSyncObj(
        input_dep, 1, &sync_obj, &sync_descr));

    if (sync_descr->type_index == HHH_CUDA_EVENT_T) {
        cudaEvent_t cuda_event = (cudaEvent_t) sync_obj;
        CudaStream cuda_stream = getCudaStreamOfHiHATTask(task);
        cudaStreamWaitEvent(cuda_stream, cuda_event, 0);
    } else {
        // fallback: note to self that before this task is executed,
        //           I must actively wait on input_dep.
    }

    // Schedule/enqueue this task.
}
```

Enqueue dependency in CUDA Runtime.
Sample 4/5: `cudaEvent_t` and `EGLSyncKHR` interop [implementation]

```c
hhInvokeCUDAImpl(..., hhActionHndl input_dep, ...) {
    hhSyncObjHndl sync_obj;
    hhSyncObjDescr *sync_descr;
    CHECK_HIHAT(hhneActionHndlGetPostdominatingSyncObj(
        input_dep, 1, &sync_obj, &sync_descr));

    if (sync_descr->type_index == HHH_CUDA_EVENT_T) {
        cudaEvent_t cuda_event = (cudaEvent_t) sync_obj;
        CudaStream cuda_stream = getCudaStreamOfHiHATTask(task);
        cudaStreamWaitEvent(cuda_stream, cuda_event, 0);
    } else if (sync_descr->type_index == EGLSyncKHR) {
        cudaEvent_t cudaEvent;
        cudaEventCreateFromEGLSync(&cuda_event, (EGLSyncKHR) sync_obj, cudaEventDefault);
        CudaStream cuda_stream = getCudaStreamOfHiHATTask(task);
        cudaStreamWaitEvent(cuda_stream, cuda_event, 0);
    } else {
        // fallback: note to self that before this task is executed,
        // I must actively wait on input_dep.
    }

    // Schedule/enqueue this task.
}
```
Sample 5/5: memory description interface [implementation]

```c
hhuInvokeFancyImpl(..., hhActionHndl input_dep, ...) {
    hhSyncObjHndl sync_obj;
    hhSyncObjDescr sync_descr;
    CHECK_HIHAT(hhneActionHndlGetPostdominatingSyncObj(action_hndl, 1, &sync_obj, &sync_descr));

    if (sync_descr.type_index == HHH_MY_SPECIAL_TYPE_THAT_I_KNOW) {
        assert(sync descr.kind == HH_TRIGGER_EQ); // debug-only
        assert(sync descr.triggered == (int64_t) 127); // again, debug only
        assert(sync descr.state_field_offset == 0); // again, debug only
        assert(sync descr.state_field_bytes == 8); // again, debug only

        while (*((volatile int64_t*) sync_obj) != 127) {
            /* we can spin now if we have nothing else to do */
        }
    } else {
        // fallback: note to self that before this task is executed,
        // I must actively wait on input_dep.
    }

    // Schedule/enqueue this task.
}
```

Potentially, do the read/wait on the device.
Questions?
Thank you