HIHAT GRAPHS

CJ Newburn, Wojtek Wasko, Stephen Jones

July 17, 2018, updated 7/24, 8/6
OUTLINE

- Graph requirements
- Why HiHAT
- Types
- APIs
- How it works
- Requested feedback

- Goal: get early feedback vs. communicate final plan of record
REQUIREMENTS

• Staging
• Hierarchy
• Dynamism
• Generality
• Trait limitations
• Operations
• Retargetability, pluggability
• Interoperability
STAGING

- A key benefit of graphs is reuse
  - Create a graph, specialize it, reuse it
  - Amortize the costs of graph creation and resource binding, per-node handling
- Three key stages: nouns in blue, verbs in green
  - **Template**: execution and memory resources may not be bound, structure evolving
    - Still target agnostic while in template form, so doesn’t use pluggable implementations
  - **Instance**: structure is fixed, parameters may still change
    - Execution and platform-internal resources bound when instantiated (template → instance)
  - **Invocation**: structure and parameters are fixed
    - Parameters passed in and data bound when invoked (instance → invocation)
- Binding
  - *Specified* in stage: exec resources @ **template**, data resources @ **instance**
  - *Occurs* and is acted upon by the verbs that transition between stages
HIERARCHY

• Three granularities
  • Overall graph - may span multiple targets or even multiple nodes
  • Instantiable subgraph - must be handlable by a single graph-handling runtime
  • Individual nodes - may be handled by any plugged-in implementation
• Distinction between instantiable and overall graphs begets hierarchy
  • Graph of subgraphs, where nodes may have associated subgraphs
• Hierarchical support goes along with support for transformation
  • Can group a set of nodes into a subgraph
  • Can replace a set of nodes with another set of nodes, e.g. decomposition, aggregation
• When nodes get dynamically added
  • 1. Prior to instantiation - part of transformation, e.g. partitioning, decomposition
    • No special handling
  • After instantiation - new work created while on execution resources
    • 2a. Continuation within the same resource constraints
    • 2b. Follow-on instantiation with new resource assignments
• 3. Nodes may also be dynamically selected
  • 3a: Within: Graph may include a superset of nodes covering multiple control flow paths
    • This may incur extra resources to cover the union (vs. sum) of all paths
  • 3b: Across: Select which of several next graphs to execute
GENERALITY

- Not tied to any given target implementation, e.g. CUDA may not be there
- Imposing no runtime-specific restrictions, e.g. file IO is ok
- User-defined implementations may be provided for any functionality
- Supports all operations, including async allocation
- Supports interactions among multiple graphs
TRAIT LIMITATIONS

- Implementation-specific support is captured in a set of architectural traits
  - Limitations on task content, e.g. file IO
  - Capabilities for memory management, e.g. ability to clear memory, thread specific
  - Limitations on structure, e.g. only one incoming dependence unless SyncAll or SyncAny
- Registered implementations declare their support for traits
  - Think of a bit vector
- Requestors declare their expected support
  - Can either trust the user to check for supported traits, or confirm support in debug mode
OPERATIONS

• Graph nodes are generic, but they have a kind and descriptor
  • Kind and descriptor in a single struct to promote consistency
  • Pointers to descriptors are type checked within a union

• Supported action kinds
  • Memory: alloc, free, memset (for convenience)
  • Invocation - available menu of primitives plus user-defined functions
  • Copy
  • Synchronization - any or all (nop that only managed dependencies)
  • Initial focus is only on low-level (common layer) primitives
RETARGETABILITY, PLUGGABILITY

- Retargetability
  - Multiple kinds of execution resources, e.g. CPU, GPU, FPGA
  - Multiple kinds of memory resources
  - Multiple underlying runtimes, e.g. CUDA, no CUDA
- Pluggability
  - Dispatch to registered implementations that are pluggable by an admin or user
  - Implementations are selected based on target and a selector
  - Selector is applicable where the functionality may be equivalent but performance may not, e.g. different kinds of allocators, different policies
INTEROPERABILITY: MULTIPLE TARGETS

- Interface seamlessly and efficiently across multiple targets
- Synchronization: Sync objects may be used natively or via programmatic interfaces
  - They may be fully described so another agent can know how to trigger or poll them
  - Or they may be tagged and used natively by implementations that know how to use them
  - Producers and consumers can interact directly vs. through an intermediary
- Memory: Metadata in DataView object manages (among other things)
  - Which memory resources data objects can migrate to or be replicated on
  - Which execution resources can have read access, write access
  - How the stored data should currently be interpreted
- Graphs: instantiable subgraphs for each of many targets, interfacing with each other
INTEROPERABILITY: GRAPHS

- Work may be partitioned across multiple graphs
  - Graphs on the same target, e.g. sequence of subgraphs
  - Different targets on the same node, e.g. for multiple runtimes on hetero resources
  - Different nodes, with a listener thread that handles remote requests on each one
- Spanning graphs
  - Dependence edges may span graphs
  - Operations may span graphs on different targets, e.g. marshal/demarshal for a copy
- HiHAT support for cross-graph interaction
  - Interface nodes added for cross-graph dependencies and as proxies for remote actions
  - Only interface nodes interact with the runtime’s remoting capabilities
INTEROPERABILITY NODE CASES

- Incoming or outgoing dependence arcs
  - Incoming: passive, unless active polling for trigger of sync object is needed
  - Outgoing: passive for dependencies, unless active triggering of sync object is needed; may be active otherwise for a remote action on another target - see below

- Cross-target or same target
  - Only need to manage scope if same target
  - If different targets, some remoting is involved: create command, marshal parameters, put in command queue, signal target side
WHY HiHAT GRAPHS

• Retargetability
  • Multiple targets
  • Nodes (actions) are inherently retarget-ready
  • Nodes can be rebound while in template stage

• Generality
  • Node or graph granularity, graphs are serializable
  • Interfaces between graphs
  • Nodes are affinitizable to execution and memory resources
  • Graph transformation
  • Trait-based implementation restrictions
TYPES

• GraphNodeKindEnum - Alloc, Free, MemSet, Copy, Invoke, SyncAll/Any, ...
• Graphs - template, instance, invocation
• Nodes - template and instance; generic wrt kind
• Descriptors - struct with kind and per-kind parameters
GRAPH STAGES

• Creation
  • Template - explicitly constructed by clients
  • Instance - created by GraphInstantiate
  • Invocation - created by GraphInvoke
  • Each of these has a handle for HiHAT; CUDA Graphs doesn’t support invocation handles

• Nodes
  • Template - needed for explicit construction
  • Instance - needed for customization of parameters
  • Invocation - apparently not needed for CUDA Graphs, but these are HiHAT actions

• Sets of nodes are only used internally
• Arrays of template or instance nodes may be obtained from queries
APIs

- Factory: GraphTemplate - Create, Destroy, Clone, Node {Add, Destroy, Move}, Edge {Add, Remove}
- Verbs: GraphInstantiate, GraphInvoke
- GraphInstanceDestroy
- Graph getters: Graph{Template, Instance} - Get[Num]RootNodes, GetCorrespondingNode
- Node getters: Graph{Template, Instance}Get - Node[Num]Dependents, Node{Exec}Descr
- GraphAggregate, GraphAddInterfaceNodes
GRAPH CONSTRUCTION

- Client owns the graph objects and is responsible for destruction
  - Explicit \((\text{hheGraphTemplateCreate})\) or implicit \((\text{hheGraphInstantiate}, \text{hheGraphInvoke})\) creation yields handle
  - \(\text{hheGraph<stage>Destroy}\) API for all three stages

- Specialization
  - \(\text{hheGraphTemplateClone}\) a graph if the original graph is to be retained
  - Modify a graph if the original graph is no longer needed

- Nodes
  - \(\text{hheGraphTemplateNodeAdd}\) to a graph that’s under construction
  - \(\text{hheGraphTemplateNodeDestroy}\) and add to a cloned graph that’s being modified
EDGES

• Edges are not an explicit object
  • Can add an edge as a pair of nodes vs. creating an edge object first
  • Is there a usage model for associating properties with edges vs. endpoints?
• Adding and removing edges
  • $hheGraphTemplate\{Add, Remove\}Edge$ - not for instance since structure fixed
  • Arguments are source and target nodes
  • Edge creation is not done upon node creation - nodes may not exist yet
  • Edges are not explicitly associated with a graph - edges may span graphs
GRAPH GETTER FUNCTIONS

• Nodes are accessible for template and instance graphs, not invocation graphs
• Nodes are associated with a graph or subgraph
  • Organized hierarchically, so # nodes is specific to a level in the hierarchy
• Nodes in a graph are accessible based on dependence-based traversal
  • Get array of up to N roots: hheGraph[Template/Instance]Get[Num]RootNodes
  • Get array of up to N dependents: hheGraph [Template/Instance]Get[Num]Dependents
NODE GETTER/SETTER FUNCTIONS

• Node-specific information is captured in two structures
  • Node description, *hheGraphNodeDescr*
    • Kind of node
    • Descriptor includes execution and memory resources - mutable during template stage
    • Descriptor includes parameters - mutable during instance stage
    • These are unique per node
  • Execution description, *hheGraphNodeExecDescr*
    • Execution policies and configuration
    • These may be common to many or all nodes

• *hheGraphTemplateGetNodeDescr* returns a pointer to the *hheGraphNodeDescr*
• *hheGraphTemplateSetNodeDescr* passing in a pointer to the *hheGraphNodeDescr*
• *hheGraphTemplateGetNodeExecDescr* returns a pointer to *hheGraphNodeExecDescr*
• *hheGraphTemplateSetNodeExecDescr* passing in pointer to *hheGraphNodeExecDescr*
AGGREGATE, MOVE

- Nominal starting point: 1 flat root graph (by convention only) with all template nodes
- Next step: aggregate subsets of nodes into subgraphs
  - Criteria: execution resources (node descriptor) or a generic tag (ExecCfg)
  - What other criteria are interesting? Connected component(s)?
  - Matching: criteria is a subset of actual, exact match, or actual is a subset of criteria?
- Optional step: shuffle nodes among subgraphs with GraphNodeMove
  - Example: partition 30 nodes into 10+10+10, then shuffle to be 10+9+11
  - Edges are unchanged upon Aggregate or NodeMove
ADDING INTERFACE NODES

• Goals
  • Make interfaces among subgraphs explicit
  • Instead of specializing all node types for interfacing, just specialize interface node
• Timing, control
  • Incremental maintenance could be wasteful - wait until needed and ready/stable
• Criteria
  • Necessary: Edge endpoints are in different subgraphs (siblings, ancestors, whatever)
  • Sufficient: May not need special handling unless in different targets
HOW IT WORKS

- Target-agnostic template nodes
- Mapping HiHAT Graphs to CUDA Graphs
- Async memory management
TARGET-AGNOSTIC TEMPLATE NODES

• Template nodes are works in progress
  • Execution and memory resources may be rebound
    • → Don’t call target-specific implementation while in template form: avoid redos
  • Created nodes will eventually be bound to at least one execution target

• Nodes vary in how target agnostic they are
  • If the functionality is supported on all targets, they are completely agnostic
  • The functionality may be supported on only a subset of targets, e.g. `calloc (clear)`
  • Parameters may be target specific, e.g. policies, `shm` sizes
  • Preregistered function handles are target specific: scheduler knows, vs. string lookup
MAPPING HIHAT GRAPHS TO CUDA GRAPHS

• First pass
  • Creation of graph, nodes, edges at template stage is mapped to directly CUDA Graphs
  • Restrictions: all and only CUDA Graphs, one (sub)graph at a time

• Second pass
  • Creation of graph, nodes, edges at template stage is handled by only HiHAT
  • Upon instantiation of a (sub)graph, template graph, nodes, edges created for CUDA Graph, then GraphInstantiate is called
  • Eased restrictions: multiple graphs, different kinds of graphs
  • Advantage: less initial work, tests directness of mapping
ASYNC MEMORY MANAGEMENT

• A target need not support full HiHAT-compatible memory management natively
  • Reference implementations could be created per target and plugged in
• Full freedom to manage alloc/free on a target or from the (CPU) host, to use sub-allocators, to access virtual addresses anywhere appropriate, on host or target
  • It’s recommended that allocs via HiHAT be paired with frees via HiHAT
• “Interpolation light” using the current implementation
  • Want to be able to dynamically allocate, provide an address of returned address, use it in a node in the same graph
  • References to allocated address can be dereferenced in a global (non-parameter) variable inside another task which is control dependent on the completion of the async alloc.
  • Standard functions like memcpy could have additional versions with parameters that are addresses of parameters vs. real parameters.
• Dealing with allocation failures
  • Success or failure of an action can be encoded in the ActionHndl’s completion status
REQUESTED FEEDBACK

• Generic nodes with kind descriptors
• Use of HiHAT types
• Need for explicit edge type
• Dependencies added separately from node creation?
• Multiple input dependences on any node or just SyncAll/SyncAny?
• Node access via dependence-based traversal; also as an unordered set?
• Nodes in graphs are managed hierarchically
**GENERIC NODES AND KIND DESCRIPTORS**

```c
typedef struct {
    hhAPINode node_type;
    Bool remote;
    union {
        hheGraphNodeAllocDescr Alloc;
        hheGraphNodeFreeDescr Free;
        hheGraphNodeMemsetDescr Memset;
        hheGraphNodeCopyDescr Copy;
        hheGraphNodeInvokeDescr Invoke;
        hheGraphNodeSyncAllDescr SyncAll;
        hheGraphNodeSyncAnyDescr SyncAny;
        hheGraphNodeSubgraphDescr Subgraph;
        hheGraphNodeSyncAllRemoteInDescr SyncAllRemoteIn;
        hheGraphNodeSyncAllRemoteOutDescr SyncAllRemoteOut;
    } descr;
} hheGraphNodeDescr;
```

```c
typedef struct {
    hhMemHndlSet mem_resrc_set;
    hhResrcHndlSet exec_resrc_set;
    hhDataView *out_mem_hndl;
    void **out_addr;
    size_t num_bytes;
    hhMemTrait mem_trait;
} hheGraphNodeAllocDescr;
```

```c
typedef struct {
    hhMemHndlSet mem_resrc_set;
    hhResrcHndlSet exec_resrc_set;
    hhDataView dst;
    size_t dst_offset;
    hhDataView src;
    size_t src_offset;
    size_t num_bytes;
} hheGraphNodeCopyDescr;
```

```c
typedef struct {
    hhGraphTemplate subgraph;
} hheGraphNodeSubgraphDescr;
```
ADDING A NODE

hhRet hheGraphTemplateNodeAdd (  
    hheGraphNodeDescr *descr; 
    // Descriptor for kind of node (NULL if nop)  
    hheGraphNodeExecDescr *exec; 
    // Descriptor struct with ExecPol, ExecCfg (NULL if not async)  
    hheGraphTemplate graph; 
    // Graph to add this node to (must not be NULL)  
    hheGraphTemplateNode *out_node 
    // Handle to created node (NULL if no handle needed)  
);  

typedef struct {  
    hhExecPol exec_pol;  
    hhExecCfg exec_cfg;  
} hheGraphNodeExecDescr;
typedef struct {
    void *chooser;                       ///< value or pointer to blob that selects among registered implementations
    size_t num_parameters;        ///< number of parameters for exec policy
    size_t parms[NUM_EXEC_POL_PARMS];///< parameters for exec policy
} hhExecPolImpl;

struct hhExecCfgImpl {
    hhExecOrderEnum exec_order;       ///< FIFO, per_dep
    hhAsyncModeEnum async_mode;       ///< Async, block, defer
    int prof_action;                  ///< One or more profiling actions
    int prof_option;                  ///< One or more items for profiling information to record
    int prof_state;                   ///< One or more ActionStates to profile
    hhExecScopeImpl exec_scope;       ///< Execution scope
    hhTagSetImpl tags;                ///< Tags (does not implicitly include the one in exec_scope)
};