HiHAT Feedback Teaser

Hierarchical Heterogeneous Asynchronous Tasking

Sean Treichler, CJ Newburn

Version 170815

TOPICS

- Going stateless
- Resource handling
- Memory abstraction and traits
- Execution scopes

GOING STATELESS

Principles

- State can lead to contended access that doesn’t scale. Avoid it.
- Implicit state tends to not be thread safe. Avoid it.

- There may be many configurations; each has its own runtime-generated handle.
- Configurations a specified for each action, vs. having a “current config.”
  - This can lead to extra parameters: execution mode, profiling mode, scope, resources
  - Changing configuration may not be free; perhaps it can be changed off of the critical path with a null action

GOING STATELESS

Example

Stateful

```
set_device(A);
f1; // on A
f2; // on A
set_device(B);
f3; // on B
f4; // on B
set_device(A);
f5; // on A
```

Stateless

```
f1(A);
f3(B);
f2(A);
f4(B);
f5(A);
```

Fewer instructions
More parameters
Richer intermixing

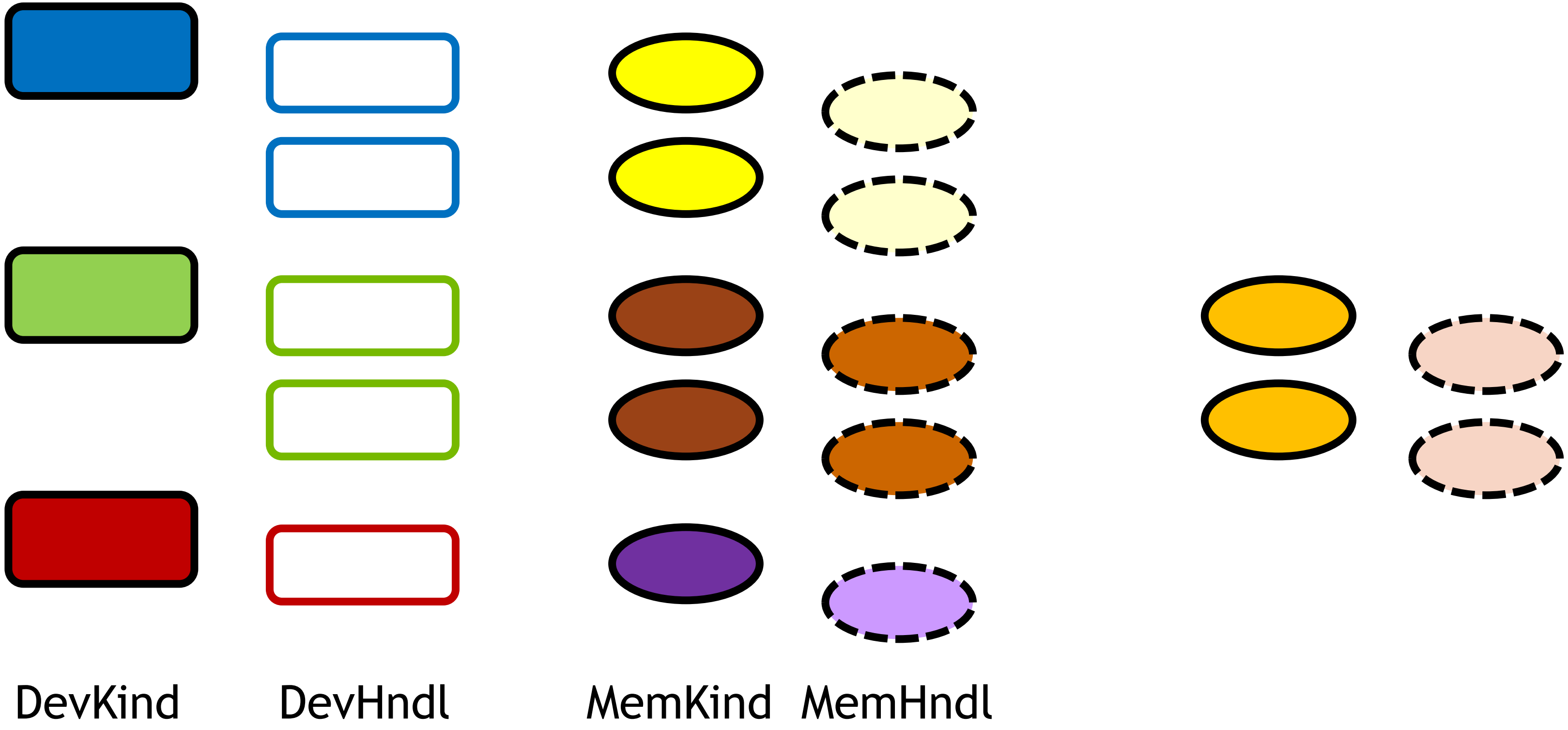
RESOURCE ENUMERATION

Goals and Expectations

- Goals
  - What’s there – enumerate it once, avoid double coverage
  - How it’s connected – number and kinds and characteristics of links
  - Cost models – access characteristics, for unloaded and shared use
- Expectations
  - Core set of basic enumerations of what’s there
  - Extended, target-specific enumeration of additional features, e.g. connectivity, costs
  - Enumeration informs cost models, cost models are specialized for each scheduler

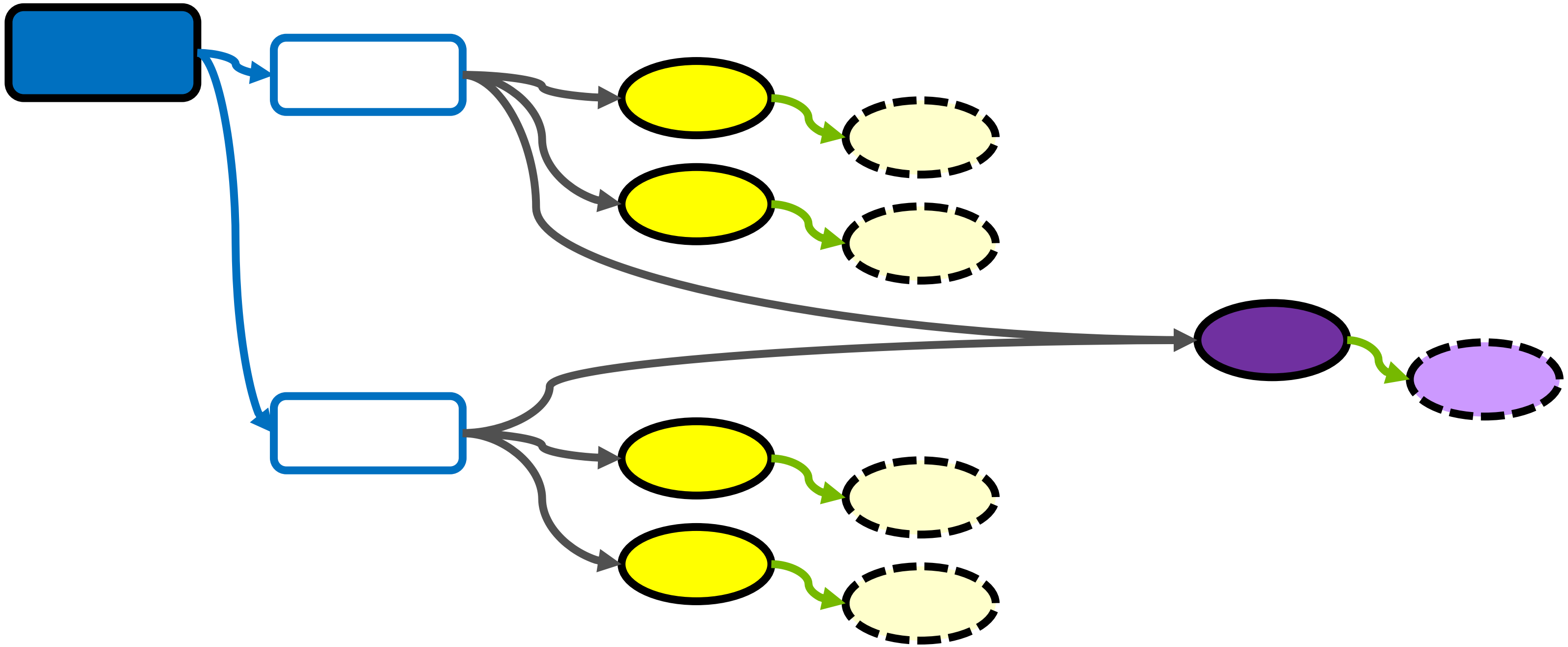
RESOURCE ENUMERATION

Device and memory hierarchy



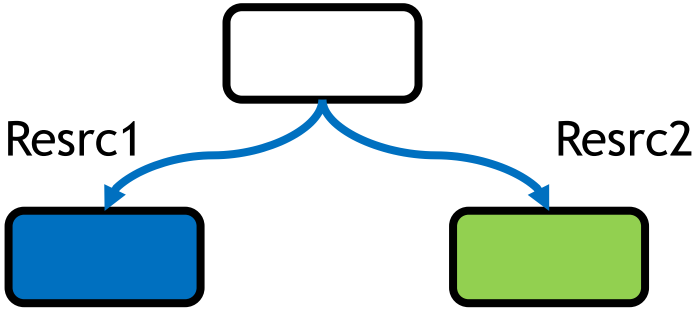
RESOURCE ENUMERATION

Both one to many and many to one



DevKind DevHndl MemKind MemHndl

RESOURCE HANDLING



```
hhRet hhnRegAPIImpl(              // register impl of HiHAT API
  void (*func_ptr)( void* ),      // function pointer
  ResrcHndle resrc_hndl,          // where this func ptr can execute
  hhAPIEnum which_api);           // which HiHAT API to implement
```

- Actions (invoke, alloc, …) are mapped to resources
  - Devices, memories and the subset of resources within those
- Clients specify resources, runtime provides a handle
- Submit action with resource handle → dispatch to implementation for that resource
  - Implementations get registered for relevant subsets of resources

MEMORY

DataView abstraction, with traits

- Program variables are represented as DataViews
- DataViews are a logical handle
  - Deferred materialization can overlap long-latency pinning, affinitization, etc.
  - Deferred allocation enables use of temporary buffers
- DataViews have a memory kind, a layout and a set of traits
  - Declarative approach supports allocation and registration, eases retargetability

MEMORY

A declarative approach

```
hhuMkMemTrait(…, HH_NVM, &mem_trait_nvm);
hhuMkMemTrait(…, HH_HBM, &mem_trait_hbm);
hhuAlloc(size, mem_trait_nvm, &data_view1, …);
hhuAlloc(size, mem_trait_hbm, &data_view2, …);
size_t offset1 = offset2 = 0;
hhuCopy(data_view2, offset2, data_view1, offset1, size, …);
```

- cudaMempyToSymbol vs. cudaMemcpy
- cudaMemcpy(dest, src, size, cudaMemcpyDeviceToHost)
- MemKind: DDR, HBM, NVM, SHMEM, CONST

EXECUTION SCOPES

Enable clients to communicate usage info to runtime

- Group actions together – support aggregation and hierarchy
    - Not tied to lexical scope, can overlap
- Enable efficiency
- Enhance productivity
- Support flexibility