A declarative approach to managing memory

CJ Newburn, Principal HPC Architect for Compute SW @ NVIDIA

OUTLINE

Part of HiHAT community project

- Portability challenges
- Declarative vs. imperative
- Abstraction
- Traits
- Data views
- Enumeration
- Results
- What's next

PORTABILITY CHALLENGES: MEMORY

Supporting ample diversity

- **Heterogeneity** of **physical memory**
  - **Device kinds**: NVM, DDR, HBM, SW managed, specialized buffers
  - **Characteristics**: speeds & feeds – asymmetry, capacity, connectivity, read only vs. writable
- **Program semantics, usages**
  - Archival vs. scratch, **temporary staging vs. enduring**
  - Streaming vs. random access, shared vs. exclusive/dirty
- **Performance tuning**
  - Data layout, affinity, pinning, …
  - Management: many pools, different allocation policies

DECLARATIVE VS. IMPERATIVE

Tease apart the roles of developer, tuner and target expert

- Declarative
  - **What, not how or when or who**
  - **Software engineering**: Can be outside of computation body
  - Can be abstracted, then **plug in best-available implementation**
- Imperative
  - **How, when, who**
  - Harder to maintain: sprinkled throughout computation
  - **Implementation is explicitly coded**

ABSTRACTION

Make increasing capabilities more accessible

- Declare
  - Runtime asked to **make it so** - up front (may be immutable) or incrementally (mutable)
  - User already made it so, just **inform the runtime** or other parts of the program
- Extensible, tunable
  - Implementations can be **plugged in for retargetability**
    - For every new memory resource, for every new memory trait
    - Includes ability to plug in **allocators**
  - Developer and tuner don't *need* to know how, but it's **transparent and controllable**
  - **Scheduler** can make use of just what's enumerated as supported on a given platform

MEMORY

Significant code changes if we decide to put `const_values` array in global memory.

```
__constant__ double const_values[100];
double fixed_value[100];        int size2 =
100*sizeof(double);
double *buffer1, *buffer2;

// allocation
cudaMalloc((void**)&buffer1, size) != cudaSuccess);
buffer2 = malloc(size);

// data transfer
cudaMemcpy(buffer1, buffer2, size,
cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(const_values, fixed_values, size2
                   cudaMemcpyHostToDevice));

<running kernels>

// data deallocation
cudaFree(buffer1);
free(buffer2);
```

What if `buffer2` should be allocated on a different device (like `Xeon PHI, FPGA`)?

What if our heterogenous system needs to do above operations at **runtime**?

Different flavors of APIs

IN HiHAT

```
// Setup code can be tailored for target resources
// Define memory spaces in our system
hhuMkMemTrait(…, HH_NVM, &mem_trait_nvm);
hhuMkMemTrait(…, HH_HBM, &mem_trait_hbm);
hhuMkMemTrait(…, HH_DRAM, &mem_trait_dram);
size_t offset1 = offset2 = 0;
trait = cuda_is_available ?
    mem_trait_hbm : mem_trait_dram;

hhuAlloc(size, mem_trait_nvm, &data_view1,
hhuAlloc(size, trait, & data_view2, …);

// Usage in computational loops is target agnostic
// No notion of memory type or device type
hhuCopy(data_view2, offset2, data_view1, offset1, size,
…);

// Free all the allocated resources on all devices
hhuClean(…);
```

Different memory kinds:
DDR, HBM, NVM, CONST

Different devices:
CPU, GPU, FPGA

User interfaces

target agnostic

HiHAT is at the boundary

Target implementations

target specific

RELATIONSHIP TO OTHER SYSTEMS

Come to memory breakout this afternoon to see layered Venn diagram

- Beneath user-facing abstractions
  - Runtimes: Kokkos, Raja, …
  - User-facing memory abstractions: Chai & Sidre on Umpire; SICM, OpenMP, libmemkind?
- Part of HiHAT project
- Implementations plug in from below
  - mmap, libnuma/numactl/mbind, hwloc, OS support, TAPIOCA, libpmem
  - cnmem, tcmalloc, jemalloc, cudaMalloc, cudaMallocManaged, …

TRAITS

Declare fundamental and accidental properties

Semantics

- Size
- Usage/access pattern: read only, writable; [random, streamed]; ld/st vs. block

Performance

- Device kinds: NVM, DDR, HBM/MCDRAM, SW managed, specialized buffers
- State: **materialized, affinitized; pinned**; valid, cleared, dirty
- Management: blocking, async, deferred; **unified**; **policies** (locality, alloc, coherence)
- Data layout: aligned, {dimensions, ranks, stride, block size, …}

DATA VIEWS

Data collection + metadata

- Logical abstraction of **program variable** with declared traits
  - Programmer cares about a set of elements in data collection, declares a few **usage hints**
  - Tuner cares about traits that influence **performance**
- Allocation
  - Pass in traits, **resources**, where address get stored; get back a handle to data view
  - Implementation invokes allocator that's registered for those resources, enforces traits
  - **Deferred materialization** can overlap long-latency pinning, affinitization, etc.
  - **Deferred allocation** enables use of temporary buffers
- Registration
  - Pass in traits, resources, address; get back a handle to data view
- Getters for all traits, setters for mutable traits (migrate, pin, relayout?, etc.)

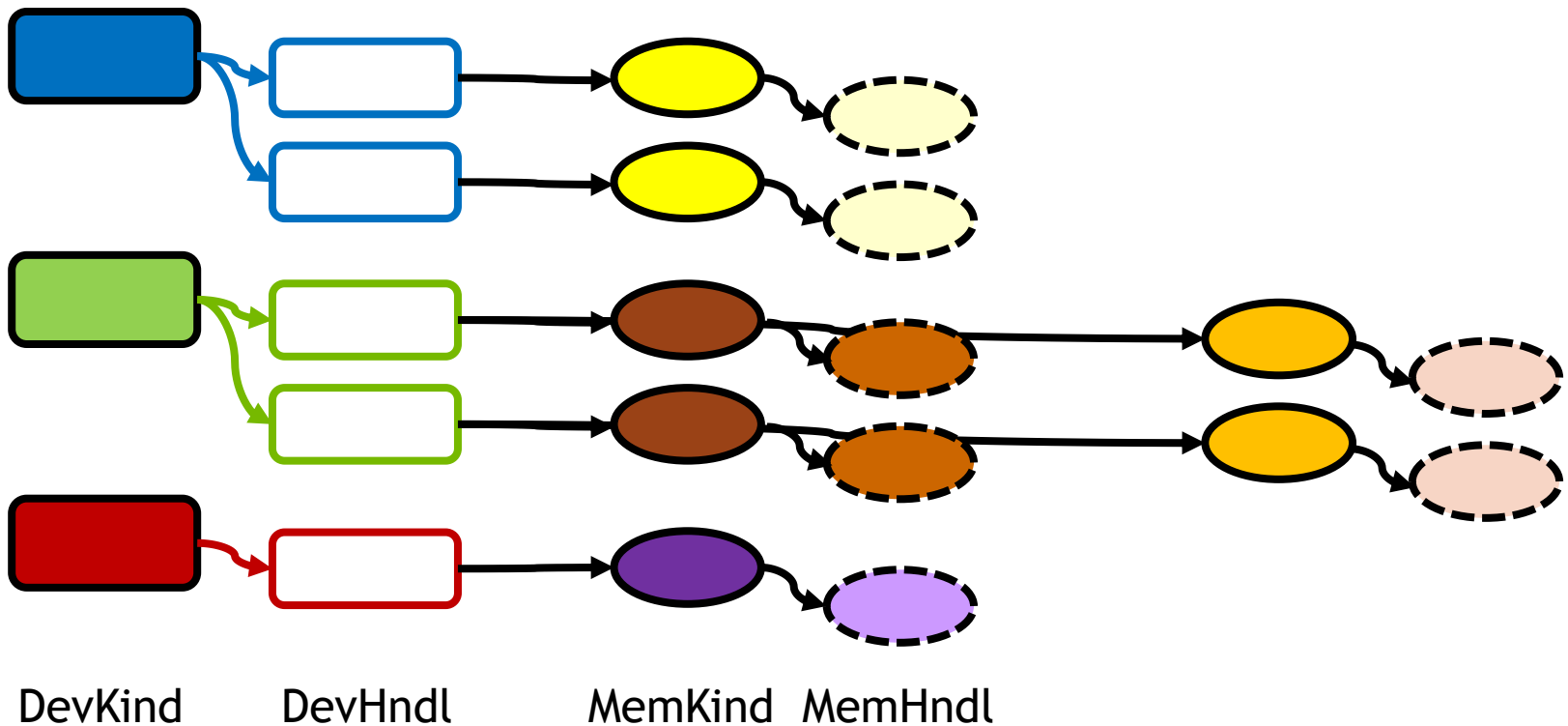
RESOURCE ENUMERATION

Goals and Expectations

- Goals
  - What's there – enumerate it once, avoid double coverage

RESOURCE ENUMERATION

Device and memory hierarchy



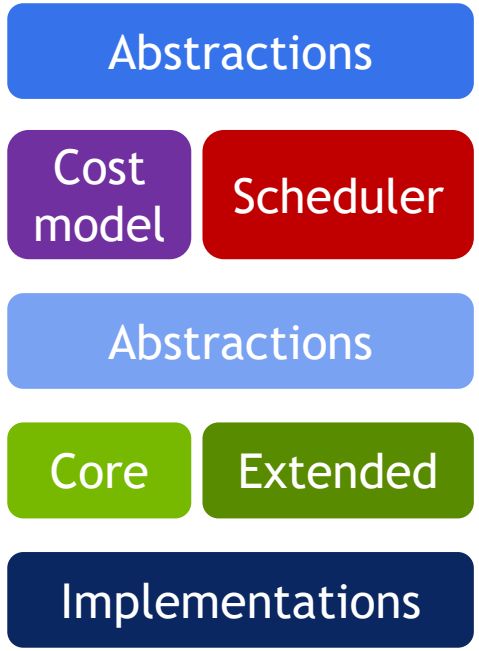
RESOURCE ENUMERATION

Goals and Expectations

- Goals
  - What's there – enumerate it once, avoid double coverage
  - How it's connected – number and kinds and characteristics of links

RESOURCE ENUMERATION

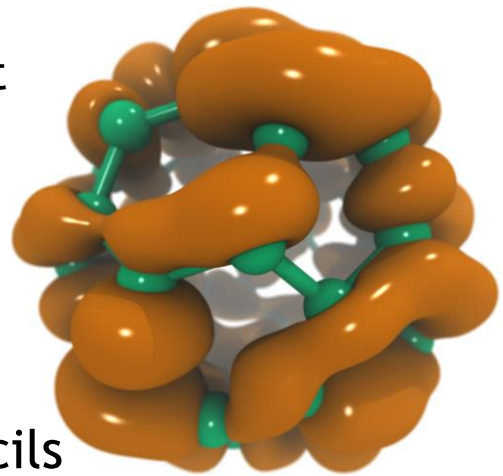
Goals and Expectations



- 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 add'l features, e.g. connectivity, costs, order
  - Enumeration informs cost models, cost models are specialized for each scheduler

MOLECULAR ORBITALS (MO) APPLICATION

- Compute wavefunction amplitudes on a grid for visualization
  - Evaluate linear combination of Gaussian contractions (polynomials) at each grid point, function of distance from atoms
- Algorithm made arithmetic bound via fast on-chip memory systems
- Three different algorithms for different memory structures:
  - GPU constant memory
  - Shared memory tiling
  - L1 global memory cache
- Representative of a variety of other grid-oriented algorithms, stencils
- Use of special GPU hardware features, APIs helped drive completeness of HiHAT proof-of-concept implementation already at an early stage



MOLECULAR ORBITALS PERFORMANCE

- Performance of MO algorithm on HiHAT User Layer PoC implementation closely tracks CUDA performance.
- Spans x86, POWER and Tegra ARM CPUs

HIHAT API GAINS FOR MOLECULAR ORBITALS APPLICATION

Molecular Orbital Algorithm, Mem Kind		Speedup vs. ShMem	HiHAT API gain
x86 + GPU	SharedMem HiHAT	1.000x	1.028x
	L1CachedGlblMem HiHAT	1.088x	1.025x
	ConstMem HiHAT	1.472x	1.031x
PWR + GPU	SharedMem HiHAT	1.000x	0.999x
	L1CachedGlblMem HiHAT	1.116x	1.001x
	ConstMem HiHAT	1.534x	0.983x
ARM + GPU	SharedMem HiHAT	1.000x	-
	L1CachedGlblMem HiHAT	1.094x	-
	ConstMem HiHAT	1.059x	-
	NoPin-SharedMem HiHAT	2.349x	0.995x
	NoPin-L1CachedGlblMem HiHAT	2.561x	0.984x
	NoPin-ConstMem HiHAT	2.562x	0.998x

PORTABILITY ON MO

Mapping between CUDA and HiHAT

- Time to port MO: 90 minutes
- HiHAT has fewer unique APIS (6 vs. 10)
- HiHAT has fewer static API calls (30 vs. 38)

- **Accelerate optimization space exploration**
- Also enhance coding productivity

TARGET-SPECIFIC API USAGE IN MOLECULAR ORBITALS APPLICATION

Category	Original CUDA		Ported to HiHAT	
Invoke	<<<>>>	3	hhuInvoke()	3
Data mvt	cudaMemcpy()	7	hhuCopy()	7
	cudaMemcpyToSymbol()	7	hhuCopy()	2
Configuration	cudaSetDeviceFlags()	1	(config)	0
	cudaFuncSetCacheConfig()	2	(config)	0
Data mgt, minimal	cudaMalloc()	7	hhuAlloc()	7
	cudaMallocHost()	1	hhuAlloc()	1
	cudaHostAlloc()	1	hhuAlloc()	1
	[free]		hhuClean()	[1]
	[symbols]	-	hhuRegMem()	7
Data mgt, eliminatable	cudaFree()	7	hhuFree()	(7)
	cudaFreeHost()	2	hhuFree()	(2)
	[symbols]	-	hhuDeregMem()	(7)
Coordination	-	0	hhuSyncAll()	1
Totals				
static	14+3+3+9+9+0	38	9+3+0+16+16+1	43
static min'l	14+3+3+9+9+0	38	9+3+0+17+0 +1	30
unique	2+1+2+5+0+0	10	1+1+0+2 +2 +1	7
unique min'l	2+1+2+5+0+0	10	1+1+0+3 +0 +1	6

WHAT'S NEXT

- Top down: Usage requirements
  - Applications, runtimes, programming models (e.g. OpenMP)
  - User-facing memory abstractions: Chai & Sidre on Umpire; SICM, OpenMP, libmemkind?
- Bottom up: Expose the goodness of available HW and SW implementations
  - mmap, libnuma/numactl/mbind, hwloc, OS support, TAPIOCA, libpmem
  - cnmem, tcmalloc, jemalloc, cudaMalloc, cudaMallocManaged, …
- Proofs of concept
  - Implement and try it out
  - Can build on top of open source HiHAT infrastructure