HiHAT and graphs

next steps in light of the CLay-GPUs workshop

Outline

CLay-GPUs (C++ Layering on GPUs) workshop summary

Workshop outputs

Next steps in the context of HiHAT

The Workshop

Hosted by CSCS at ETH Zurich. Two days. Topics covered:

- Infrastructure: CUDA Graphs/HiHAT/C++ Executors/C++ Futures/C++ Affinity
- Layering: FleCSI/Legion/Realm, HPX/Phylanx, SYCL
- Applications overview by: CSCS, FAU, ETHZ, Sandia, ORNL, LANL
- Frameworks: GRIDTOOLS, PaRSEC, Raja, Kokkos, HAGGLE, DARMA, DAPP, SLATE, compilers

Clay-GPUs workshop points of agreement

Graphs are an abstraction of interest.

It looks like graphs can be built up using the (revised) proposals for executors and futures. We need to work through examples to build confidence in this. Different executors will be necessary for static building or dynamic building of graphs.

CUDA Graphs look interesting enough to try.

Of particular interest are lowering overheads in support of fine granularity, graph reuse, graphs with control flow in them. We should collaborate on creating a set of proofs of concept implementations of executors in support of CUDA Graphs.

Some runtimes implement primitives for several targets and may benefit from HiHAT.

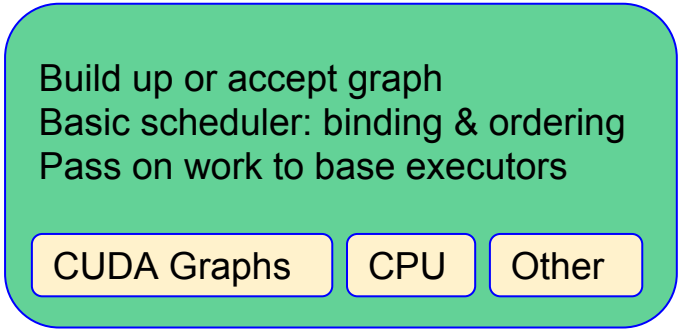
They can try out HiHAT to see if it provides ease of use, simplicity, performance and robustness. Those interested can “spend a day” identifying a candidate use, studying the API doc and doing a trial run with HiHAT.

See here for detailed notes from the workshop.

Possible layering

C++ code
  Pass work to hetero executor
  Either one task at a time,
    or as a (reusable) graph

  Hierarchy of executors:
    Hetero → base



Code in hetero executor could be specialized for each base executor, or each base executor could be wrapped in a common interface, with implementations plugged in below

Workshop follow up action items

Develop a set of **parallel pattern building blocks** that are key for influencing the standard

Collaborate on creating **POC implementations of executors** in support of CUDA Graphs

C++ interfaces for expressing **lazy execution and building graphs**

Create a set of **APIs for building up a graph**

Enumerate usage models for **dynamic task creation**

HiHAT - investigate binding of events between Fortran/C++/other as a less complex interaction mechanism

Provide CUDA Graphs documentation

External dependencies (CPU/GPU signalling nodes), and thread management for CPU tasks.

Graphs

CUDA Graphs

Where it helps, how it's used

- Potential benefits
  - Reducing overheads for small tasks, repeated graphs
  - Resource management (cache locality), dynamic control flow, CPU-less interop
  - Improved GPU utilization - over execution & memory resources
- Staged progression
  - Build as **template** with deps → **instantiate** with bindings → **invoke** with parameters
  - Instantiation and invocation are only supported for CUDA Graphs, nothing else yet
    - Can instantiate+launch as a single runtime step:
      - lose repeat-launch benefits (i.e. launch operation is slower than streams)
      - still gain utilization & execution-overhead benefits (may be worth slower launch)
    - → Therefore runtimes without instantiation concept can still benefit

Workload characterization

Based on CUDA Graphs checklist/survey

Task	Range of possible task latencies (us)	1-1M
	Functions directly access global memory vs. just using parameters	No global: H: 7, M: 2, L: 3 Global: H: 3, M: 3, L: 6
Graph	Range in # nodes in each graph	1-10K
	Range in # concurrent graphs in each MPI rank	1-32
	Expected reuse of graphs	H: 8, M: 5, L: 0
	Use of control flow	H: 8, M: 3, L: 1

CUDA Graphs Alignment

Based on CUDA Graphs checklist/survey

Natural expression	H: 2 , M: 3, L: 5
Interoperable spanning of resources	H: 10, M: 5, L: 0
Other actions, e.g. memory mgt, data mov't, sync	H: 11, M: 1, L: 1
Explicit dependencies	H: 4, M: 1, L: 3
Operand descriptions	H: 4, M: 0, L: 3
Static vs. dynamic graphs	static: 7, static/control: 12, dynamic: 9

Sequence

Larger usage context

Build program graph

- Refactor program into tasks that will be compiled for each target
- Either explicitly specify dependencies or describe operands to infer dependencies

Schedule

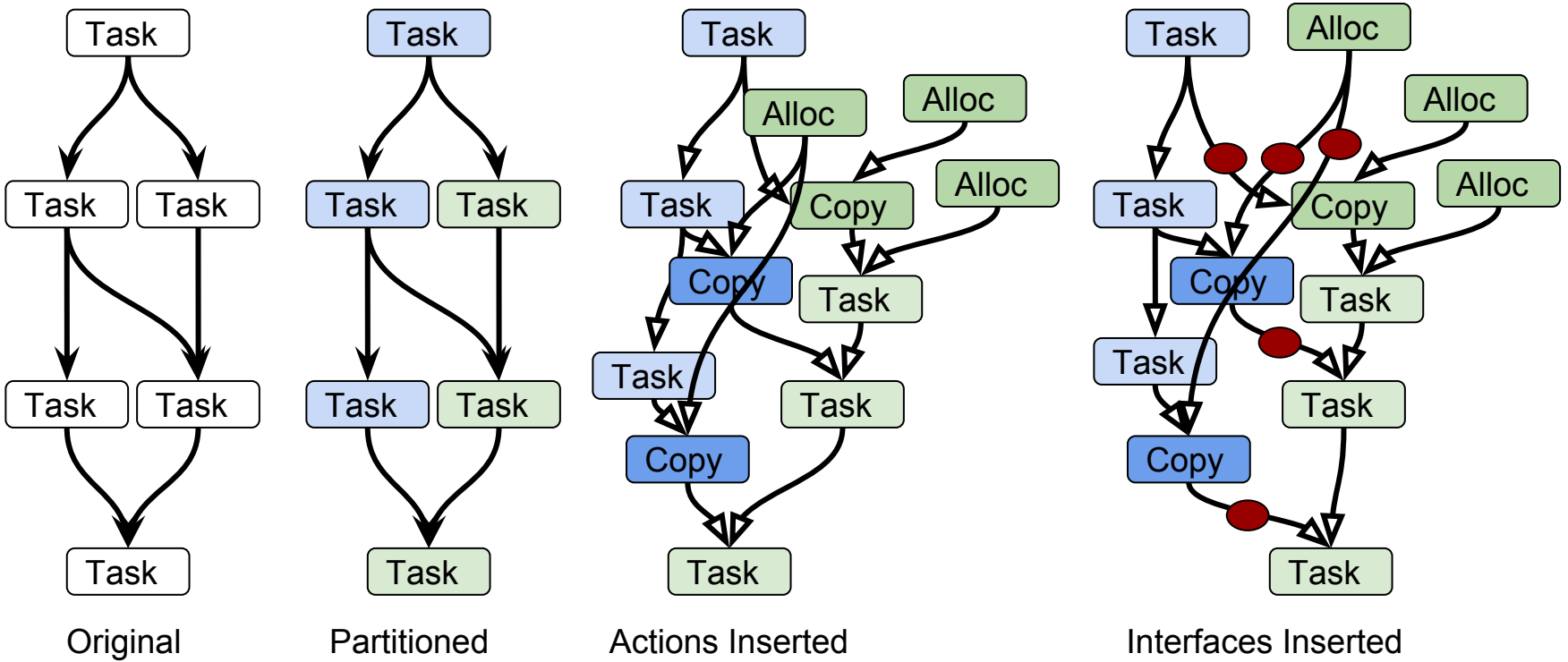
- Partition into subgraphs, e.g. by target
- Bind vertices to resources, order them
- As needed, add vertices (actions), e.g. memory mgt, copies, cross-graph interfaces

Present program graph

- Feed whole subgraph or individual vertices (lambdas and futures) into an executor

Sequence illustration

Graph → subgraphs → augmented subgraphs



HiHAT and Graphs

Why wrap CUDA Graphs in HiHAT

CUDA Graphs

- Targets: {CPU, GPU}
- Some current (temporary?) limitations
  - Restrictions on what can be executed on CPU
  - No affinity controls
- Create template, instantiate, invoke

Gaps left for HiHAT ➜ evaluating value in the context of real applications

- More targets, e.g. no GPU ➜ no CUDA, DLA, …, other
- No restrictions for what runs on CPU, affinity controls are available
- Retargetable hhAction provides portability
- Support (including serialization) for targets which don’t natively support work expressed as a graph
- Given multiple targets for graphs, there’s a need for managing interaction among graphs

Graphs on multiple resource sets

Spanning, decomposition

- Resources
  - Target – subset of a given kind of execution resource (e.g. subset of threads on CPU)
  - Execution resource set – arbitrary set of possibly-hetero compute targets {CPU,GPUs,FPGAs}
  - Graph resource set – execution resource set supported by pluggable graph implementations for GraphInstantiate, GraphInvoke: CUDA Graphs speaks {CPU,GPUs}
- A graph represents work and has a set of nodes. It's represented in stages.
  - While graph is in the template stage, it may span graph resource sets
  - Graphs in the template stage need to get decomposed into subgraphs before invocation
  - While graph is in {instance, invocation} stages, it is bound to a specific graph resource set and cannot span multiple graph resource sets

Partitioning graphs based on resource sets

- Instantiation and invocation are performed by plugged-in implementations
  - Specific to either just one resource (DLA) or collection of resources (CUDA: CPUs + GPUs)
  - If you hand-off something with DLAs to CUDA Graphs it won't understand
- Partitioning must occur at a higher level prior to hand-off. Client runtime…
  - … must prep for cross-partition interaction as needed (e.g. create extra interface nodes)
  - … may choose to create graphs for each of several targets for the same work
    - Either instantiate from multiple template graphs or instantiate, morph template, instantiate…
- Not all targets will support graph handling
  - If unsupported, instantiation/invocation will never occur on graphs
  - The original template form of nodes in the partition could be directly invoked as hhActions
    - Client runtime queries HiHAT whether resource set supports graphs, and if not, client runtime would invoke the work in the template graph at the granularity of individual nodes
    - Today's HiHAT would support this usage directly, other than for the template form

Next steps: HiHAT

Draft and peer-review a set of generic graph work submission APIs.

Design graph-based interface based on HiHAT between CUDA Graphs and other implementations.

Design implementation for non-graph targets.

“Notes to self”

- Don’t build a new graph handling library
  - Maybe not even use a graph library from HiHAT - instead, use Boost Graph Library in plugged-in implementations for e.g. graph partitioning or traversal