HiHAT: A New Way Forward
for Hierarchical Heterogeneous Asynchronous Tasking

CJ Newburn

TRENDS

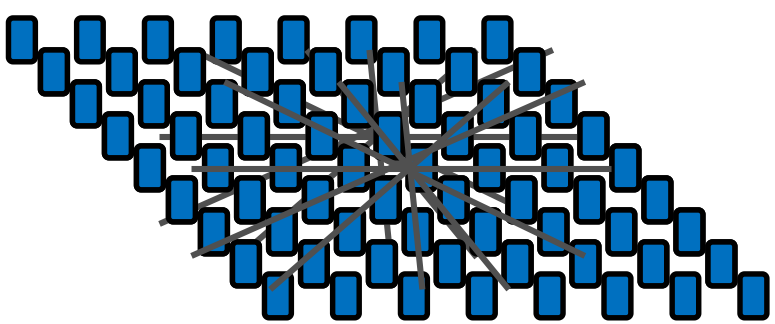
Relevant to small or large scale HPC, AI

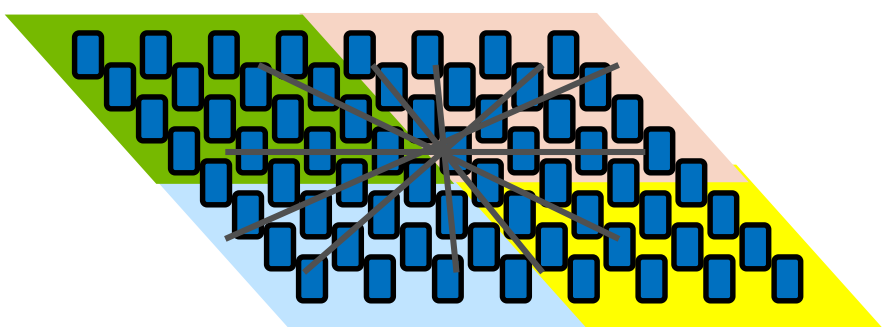
- Scale → **Hi**erarchical
- Differentiation for efficiency → **H**eterogeneity
- Unpredictability → **A**synchronous
- Functional and data parallelism → **T**asking

TRENDS

Relevant to small or large scale HPC, AI

- Scale → Hierarchical
  - Locality: higher effective bandwidth, lower latency, better TLBs
  - Abstractions that are repeatable at various levels and granularities

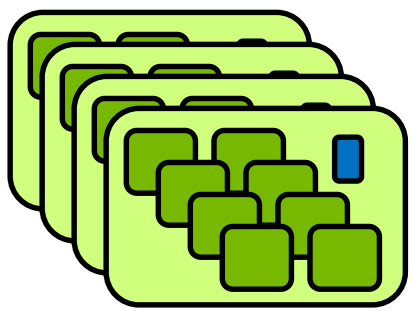




TRENDS

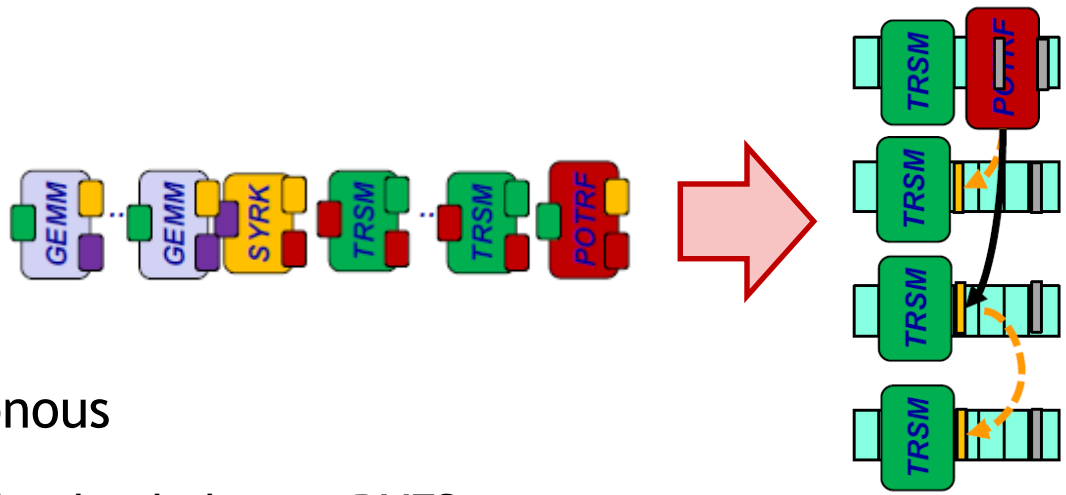
Relevant to small or large scale HPC, AI

- Differentiation for efficiency → Heterogeneity
  - Throughput and latency cores
  - Power efficiency
  - Higher aggregate bandwidth



TRENDS

Relevant to small or large scale HPC, AI



- Unpredictability → **A**synchronous
  - Varied progress: dynamic load imbalance, DVFS
  - Network congestion
  - Depth in memory hierarchy
  - ➢ *Bind and order actions from a queue onto resources with dynamic scheduling*

TRENDS

Relevant to small or large scale HPC, AI

- Functional and data parallelism → Tasking
  - Enqueue(Name, <Operands>, <Optional descriptors>)
  - Transformations: decompose, aggregate, substitute

TRENDS

Relevant to small or large scale HPC, AI

- Scale → **Hi**erarchical
- Differentiation for efficiency → **H**eterogeneity
- Unpredictability → **A**synchronous
- Functional and data parallelism → **T**asking

HiHAT

THE CHALLENGE

Widespread participation, longevity

- Meet key provisioning needs → more than a toy
  - Retargetable, library friendly, C ABI, interoperable, incremental
- Be relevant to a large market → support usages on many user interfaces
  - C++, Python, Fortran language runtimes; layered tasking frameworks; …
- Enable broad customization → open source project
  - Shared investment in pluggable building blocks, services, transformations

OUTLINE

- Past approaches
- The challenge
- A new way forward
- Value
- Context
- Momentum
- Call to action

PAST APPROACHES

"We're done with science experiments and want something we can use"

- Academic → not product quality, narrow applicability for proof of concept
- All or nothing → hard to get started, applicable only to small codes
- Limited scope → not interoperable with MPI and IO, must own main

A NEW WAY FORWARD

- Top down: community driven
  - Gather usage models, requirements, apps
  - Build momentum and interest
  - Allow for wide variety of interests
  - Consensus is a non goal – wear many hats
- Bottom up: vendor driven
  - Expose key platform features in a retargetable way
  - Connect the dots from top-down requirements
  - Assure extensible architecture; prioritize according to application priorities

PORTABILITY, RETARGETABILITY

- Portable: code doesn't have to change across targets
- Retargetable: equivalent functionality is available; transformations may be applied by human tuner, or auto-tuning or automated machine-model-based heuristics
- Functional portability is achieved by expressing semantics (the "what") cleanly
- Performance portability is achieved by abstracting the how to target-agnostic heuristics that are informed by target-specific parameters

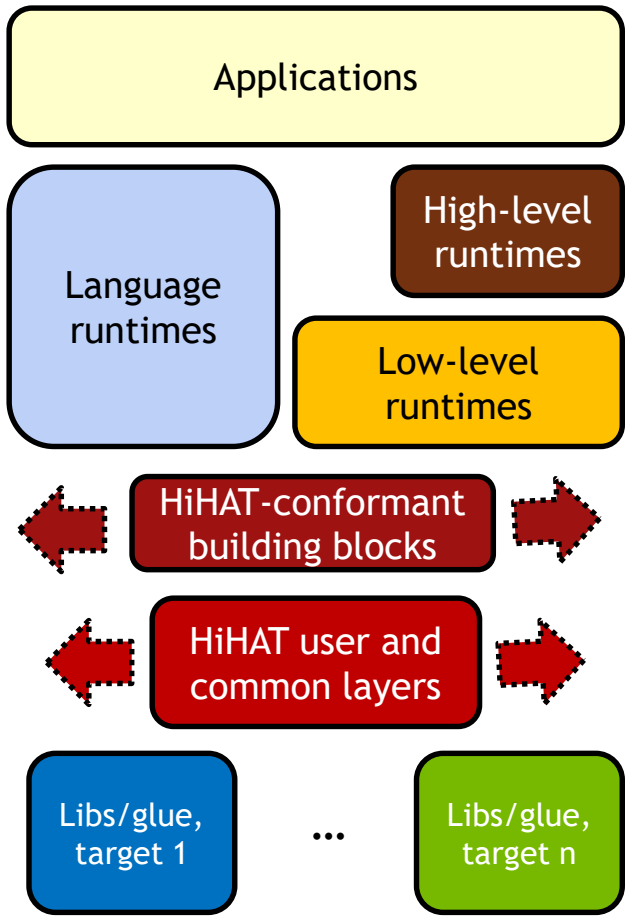
→ Separate SW into

  - Above HiHAT
    - what's not target specific, even if it's informed by target parameters → perf portability
    - what's responsible for functionality
  - Below HiHAT
    - what's target specific
    - what's responsible for target-specific performance

WHAT IS HiHAT?

4 faces

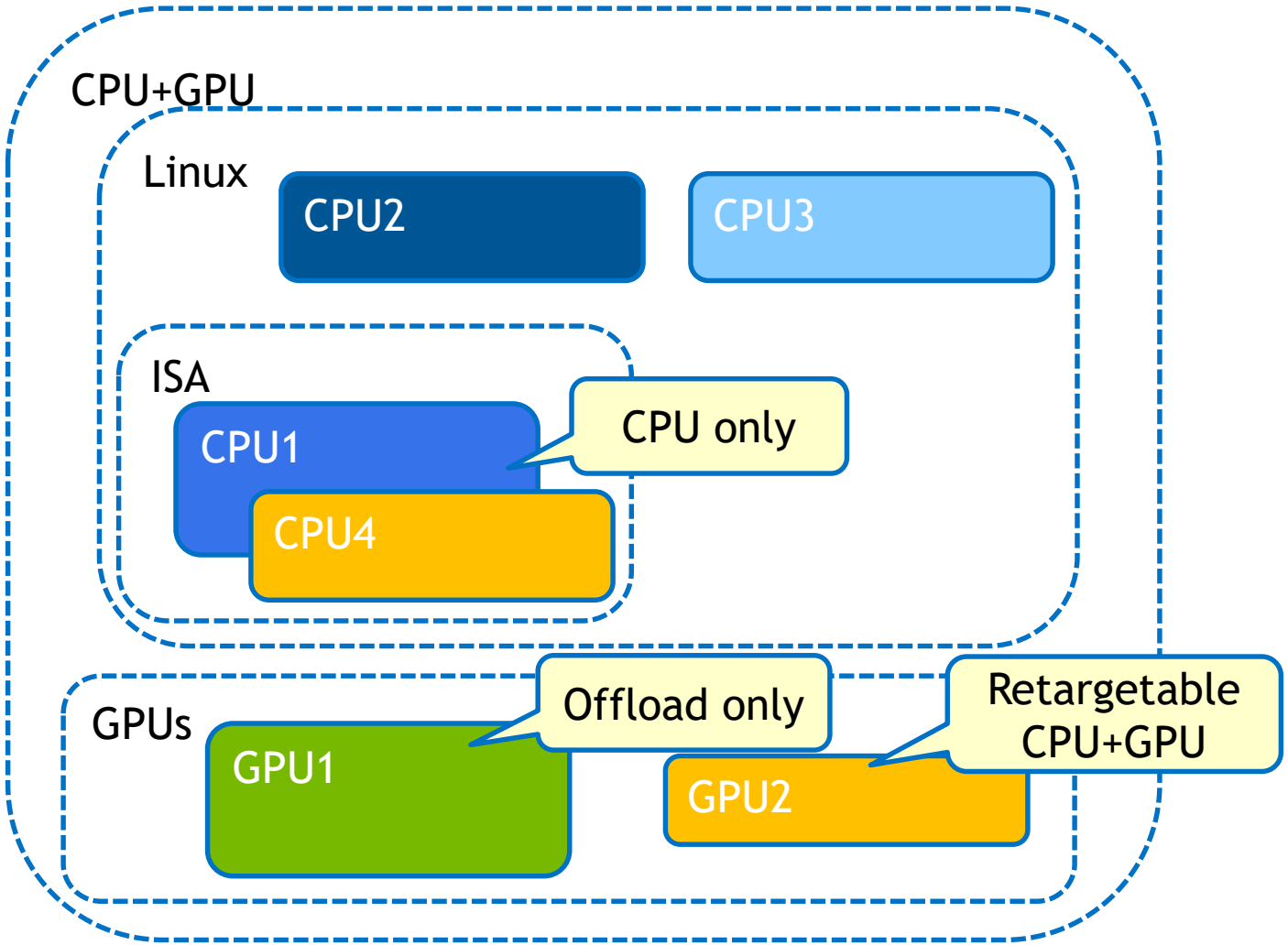
- Community-wide **requirements gathering** effort
  - Leads to solid architecture that's durable, extensible, robust
- User layer and common layer **API and implementation**
- Open source project: pluggable, conformant **building blocks**
  - Built on user and common layers
  - Language and tasking runtimes are built out of these
- **Implementation beneath** user and common layers
  - Vendor-maintained and user-supplemented



VALUE

- Common interface to vendor-specific features
  - Modular design, separation of concerns
    - What's above user/common layer can use target-agnostic heuristics on target-specific parameters
  - Future proofing
    - Retargetable across vendors, implementations, generations
    - Underlying implementations can chase changes and improvements
- Performance and robustness
  - Vendors are incentivized to provide 1st-class support; others can supplement

GROWING THE RELEVANCE PIE



SCOPE OF FUNCTIONALITY

Cover key platform-specific actions and services

- Data movement – target-optimized copies, DMA, networking
- Data management – support many kinds and layers of memory, specialized pools
- Synchronization and communication – completion events, locks, queues, collectives, iterative patterns
- Compute – target-optimized tasks, including remote invocation
- Enumeration – kinds and number of resources (compute, memory), topologies
- Feedback – profiling, load
- Tools – tracing, callbacks, pausing, … {debugging}

INCREMENTAL

- Identify what's of greatest value, e.g. for future proofing, ease, robustness
- Incrementally adopt those parts of HiHAT, and build up and out from there
- Initial target "customers" are runtimes and frameworks, rather than end users

LAYERING

Runtime, e.g. TensorRT, Legion, Kokkos, PaRSEC, Raja, C++ runtime, offload runtime

- Target-agnostic implementation that may use target-specific info
- Implemented by tuner
- Make decisions, apply transformations, call services

Reusable modules, e.g. dependence analysis, cost models, scheduler

- Target-agnostic implementation that may use target-specific info
- Implemented by tuners, open sourced
- Any kind of service that is commonly used and/or sharable

User layer, e.g. configuration, data movem't(logical source, log dest, size, layout), data mgt, invocation, sync

- Map from target-neutral API to target-specific implementation
- Implemented by target ninjas
- Some decisions, can take longer, some overhead

Common layer, data movem't(source virtual address, dest VA, DMA/memcpy), data mgt, invocation, sync

- Map from target-neutral API to target-specific implementation
- Implemented by target ninjas
- No decisions, absolutely minimal overhead

USER AND COMMON LAYER DIFFERENCES

- HiHAT User Layer – logical to low-level mapping
  - Sample inputs for higher-level and configuration actions
    - < logical source, logical target, size, [descriptor,] completion event> or
    - <func_name, logical operands, input deps, completion event, flavor>
  - Outputs
    - Low-level operands: domain, low-level addresses
- HiHAT Thin Common Layer - function mapping only
  - Sample inputs for low-level operational actions
    - < Low-level operands, size, type, completion event> or
    - <func_name, low-level operands, input deps, completion event, flavor>
  - Output: best-available implementation for that source [and target] domain
  - Razor thin, minimal overhead, no decisions
  - Provide completion events

COMMON LAYER - THIN AND LIGHT

Many possible 3rd-party implementations to select from

Function	CPU	GPU
Compute, threading	pthreads, OpenMP, Argobots, Qthreads	cu* library calls, CUDA kernels, OpenACC kernels
Data movement	MPI, SHMEM, UCX, memcpy, DMA, GASnet	MPI/GPUDirect, nvSHMEM, cudaMemcpy, DMA, GASnet
Synchronization and communication	MPI wait, MPI collectives	MPI collectives, NCCL, cudaEvent, …
Data management	malloc, TBBmalloc, new, sbrk, mmap	cudaMalloc, cudaMallocManaged, {special pools}
Enumeration	# cores, threads/core, ISA versions, hwloc, …	# devices, #SMs, compute version, topology, …
Feedback	PAPI	PAPI, cupti
Tools	Tracing? Callbacks?	Tracing? Callbacks?

USER LAYER – THICKER AND RICHER

Some of these may set up later calls to the user or common layer

Function	CPU	GPU
Compute, threading	Create OpenMP hot team, affinitize threads	Set default device
Data movement	Choose transport mechanism, given endpoints and size	Choose transport mechanism, given endpoints and size
Synchronization and communication		
Data management	Choose mem kind, allocator	Choose whether managed memory or not, choose cudaMemAdvise parameters
Enumeration		
Feedback	Load indication	Load indication
Tools	Debugging	Debugging, pause?

SERVICES

Target-agnostic pluggable services

- Build dependences
  - Convert sequence of functions into dependent tasks, or
  - Accept DAG spec
- Monitoring
  - Insert timing primitives, insert primitives that trace where & when things happen
- Visualization
  - Use enumeration to build time vs. resource matrix
  - Post-process monitoring primitive results to build event timelines
  - Show the annotated results

TRANSFORMATIONS

Pluggable operators that substitute M new actions for N old actions

- Aggregation
  - M < N, e.g. contiguous data movement, sub-sequence of tasks on same resources
- Decomposition
  - M > N, e.g. tiling, apply hierarchical refinement
- Specialization
  - Specialize the task implementation for a given memory kind or data layout
  - Manage temporary buffers: task ← moved input operands ← allocated temp buffer ← free space for move ← completed task

FUNCTIONAL BUILDING BLOCKS

Pluggable modules

- Compute costs
  - Simple: based on operand sizes, floating point arithmetic intensity factor
  - Richer: O() complexity in operand size, may depend on data layout
- Communication costs
  - Simple: based on operand size, model of bandwidth and latency for topology
  - Richer: based on data layout, e.g. contiguity, non-unit stride, whether blocked
- Scheduler
  - Simple: Earliest completion time, given data movement and compute
  - Richer: Trade off among implementations on different computing resources and with different data layouts, considering the extra costs of data re-layout

HIERARCHICAL INVOCATION EXAMPLE

- Input: sequence of function calls with operands and operand descriptors
- Root layer of hierarchy: distribute work across nodes in sub-cluster
  - Dependence analysis: discover deps among function calls; allow multiple granularities
  - Model costs: each function on each node, each data xfer between nodes
  - Convert: func → <sync on preds, move input opnds, alloc output buf, task, trigger sync>
  - Schedule: bind to nodes and preliminary order based on cost models
  - Pass down hierarchy to nodes
- Leaf layer of hierarchy: distribute work across {CPU, GPUs} resources in node
  - Configure: potentially partition resources, define # of streams
  - Model costs: each function on each {CPU, GPU}, data to/from {CPUs, GPUs}
  - Model parallelism: consider available resources and available parallelism
  - Transform: decompose appropriately, compute → <data re-layout, spcl compute>
  - Schedule: bind to {CPU, GPUs} streams, order within each stream, add alloc & sync
  - Pass sequence of {compute, data movem't, data alloc, sync} actions to HiHAT User Layer

HIERARCHY PROPOSAL

- Runtimes have a choice:
  - Span all of topological hierarchy, introduce recursive layers only for nested tasking (Legion)
  - Common SW architecture/interfaces are repeated for each topological layer (hStreams)
- Similar functionality at multiple levels of hierarchy
  - Principle of subsidiarity: make decisions as local as possible, subdivide work ASAP
  - Relevant to multiple layers in topology: transform, schedule; also load balance, fault tol.
  - Resource (compute, memory) binding can be abstract for interior of tree, specific @ leaves
- Common and user layers
  - Used at all layers of the hierarchy to do actual invocation, data movement, etc.
  - May have more-abstract analogous interfaces further up in the hierarchy

DATA MOVEMENT EXAMPLE

Resolving the abstraction as you get close to the metal

- Input: Move a collection of 5K blocks of various sizes from {CPU, GPUs} to {CPU, GPUs}
- Aggregate: Bundle contiguous chunks to same target → fewer, larger chunks
- User layer <source, target, size>
  - Instance resolution*: find closest, latest copy of source; find target affinity
  - Alias detection*: nop-ify when source & target are aliased, but maintain transitive deps
  - Pick transport type: above size threshold → DMA ops, below threshold → memcpy ops
  - Pick transport type: best RDMA implementation for end points
  - Address mapping: adjust source/target addresses by appropriate offsets for their domain
- Common layer <source domain, source adr, target domain, target adr, size, type>
  - DMA: Invoke DMA on CPU or GPU, or RDMA to remote CPU/GPU
  - Memcpy: T-threaded memcpy for T-thread targets, cudaMemcpy

*May be done above user layer

STATUS

Gradual start, but on firm footing

Gather

Usage models, applications, user requirements – modestly-broad participation, need more

Architect

Design principles – good progress, much more to come; need more concrete requirements

Implement

Implementation plan – POC this summer, anticipating partial implementation end of 2017

Integrate

Proof of concept → early adopters → broaden

CONTEXT

Wearing many hats

- Language runtimes: C++, Fortran, Python; HPX; SyCL
- Spectrum of static (deep learning frameworks) to dynamic (unpredictable imbalance)
- Plumbing under runtime frameworks: Kokkos, Raja, PaRSEC, Realm, Sandia Task-DAG
- High-level frameworks: DARMA, Legion, OCR, NVIDIA deep learning and inference, UINTAH, IBM, FleCSI
- Platform-specific libraries called: QThreads, Argobots, libnuma, libmemkind, UCX, libmpi, libfabrics, …

This list is aspirational

STATIC OR DYNAMIC

Both need a common infrastructure

- Commonalities between static and dynamic
  - Same actions: cost models, binding, ordering, allocation, data copies
  - Either can be greedy, look at a limited scope, or buffer to maximize the scope
- Similar principles, slightly different approach
  - Static vs. dynamic: make decisions, either record them for later or execute immediately
- The same (library) primitives are applicable to both
  - In order to be applicable to dynamic runtimes, can't be *only* a compiler
  - But library interfaces need to be vetted to address compiler effectiveness and efficiency

MOMENTUM

Building interest, firming up investment

- Modelado.org – neutral zone, posting of usages, requirements, apps; monthly mtgs
- Active bottom-up discussions with vendors → initial POC with glue code
- Existence proofs and past learning: hetero streams, REALM, ~OCR
- ECP – ATDM funding, PathForward2 SW, CORAL/APEX/ECP app owners from ORNL, ANL, LBL, LANL
- PASC – interest from Platform for Advanced Scientific Computing, Switzerland
- Workshop on Exascale SW Technologies (WEST) – panelist, Feb. 22
- Workshop at GPU Tech Conference – May 9 am, share progress, deepen investment
- Possible talk @ IWOCL workshop, Distributed and Hetero Programming for C/C++17
- Performance portability workshop – August
- Possible SC17 panel

CALL TO ACTION

Forging the way forward together

- Identify and prioritize opportunities to leverage HiHAT by many runtime frameworks
  - Look at amenability for changing frameworks, what interface requirements are
  - Evaluate incremental adoption of subsets of HiHAT functionality
- Identify vendor-specific features and services to expose
- Review low-level plumbing interfaces, make plans regarding support
  - Consider leveraging https://01.org/hetero-streams-library
- Contribute to HiHAT effort
  - https://wiki.modelado.org/Hierarchical_Heterogeneous_Asynchronous_Tasking
  - Join monthly calls, contribute to wiki

SUPPLEMENTAL MATERIAL

- Inspired by the MPI success story
- Task graph optimization example

GOALS, FROM SECTION 1.1 OF MPI SPEC

Inspired by a success story

Fundamental to the environment

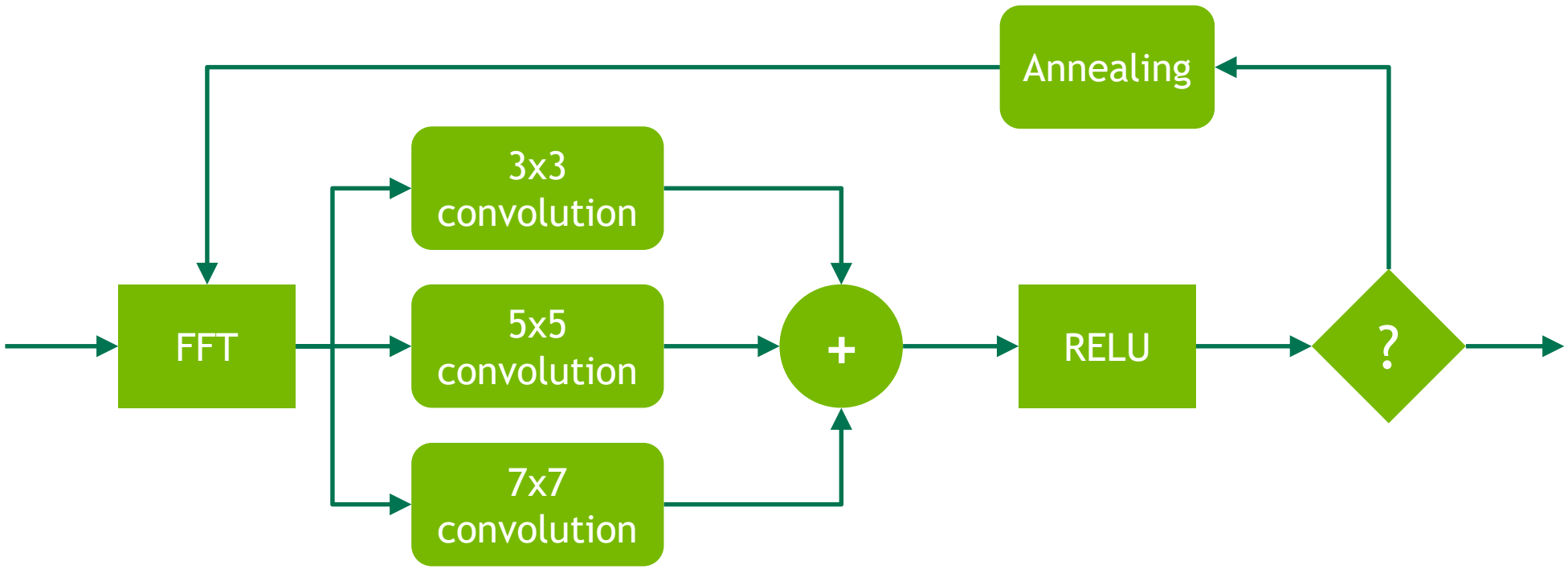
- API: library, not a language
- Heterogeneous environment: portable, easy to use
- Retargetable to many vendor platforms: clear and common interface
- Convenient C and Fortran bindings, language-independent semantics

Part of the soul of MPI, also relevant to HiHAT

- Efficient communication: enable distributed systems
- Reliable communications interface
- Thread safe

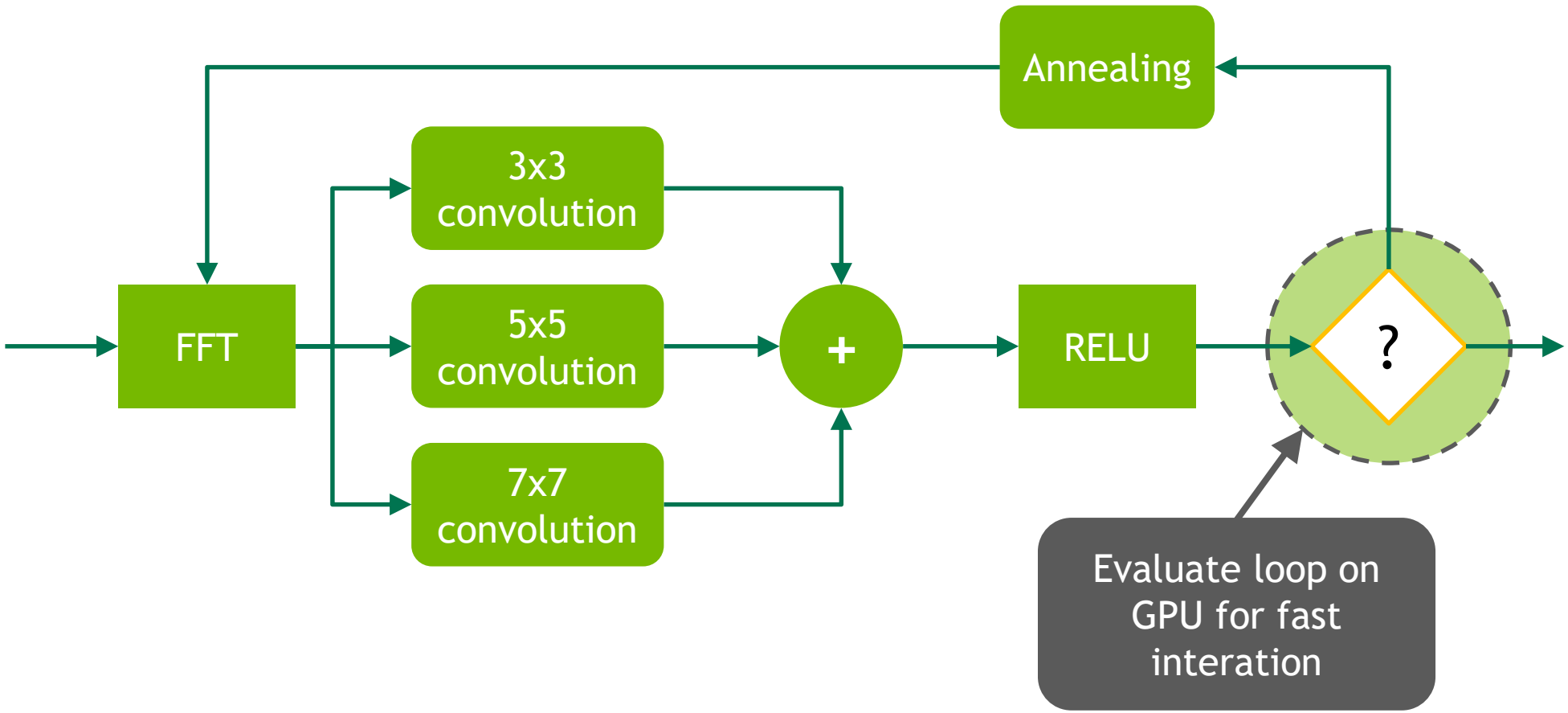
TASK-GRAPH OPTIMIZATION EXAMPLES

EXAMPLE DNN TRAINING WORK GRAPH

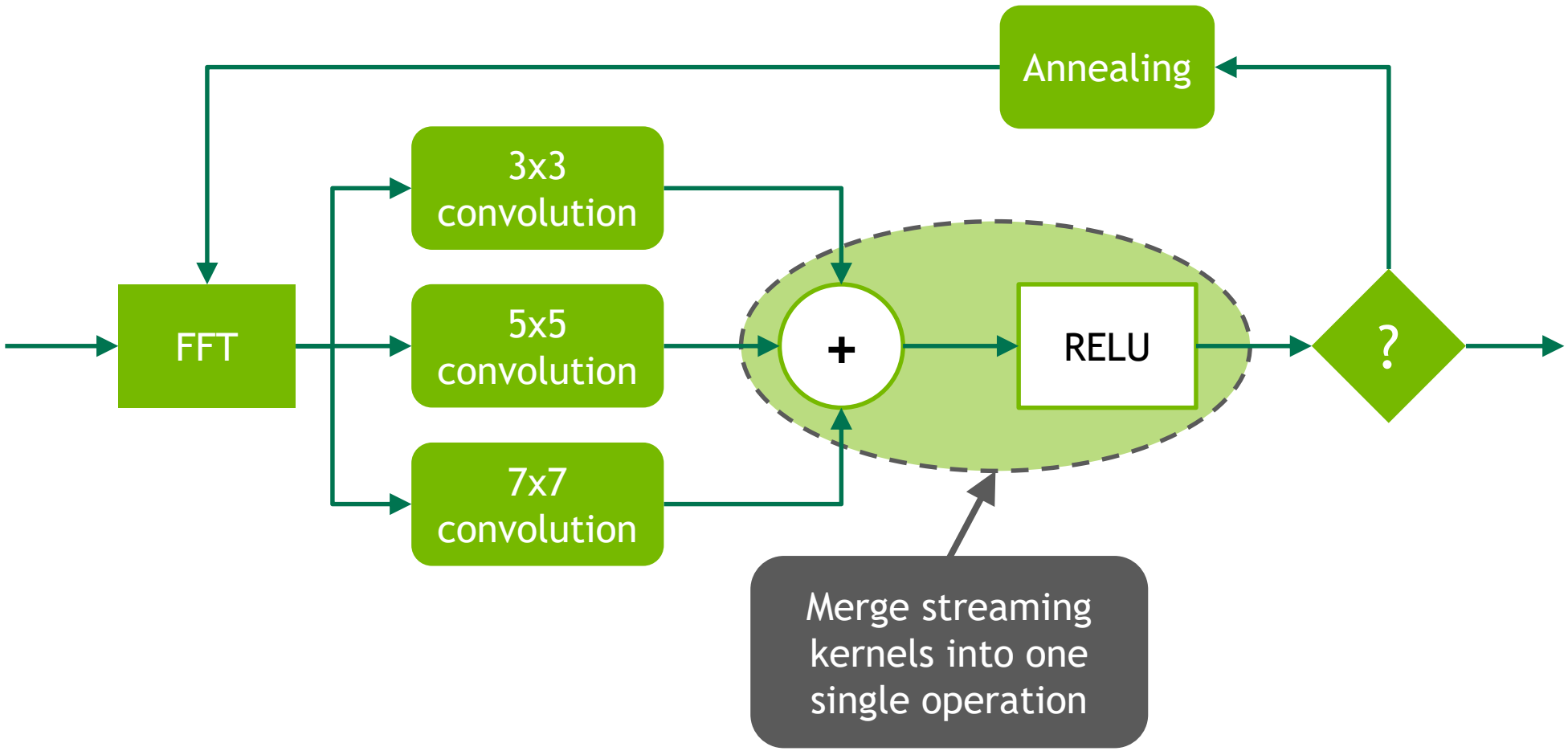


(representation only – not complete)

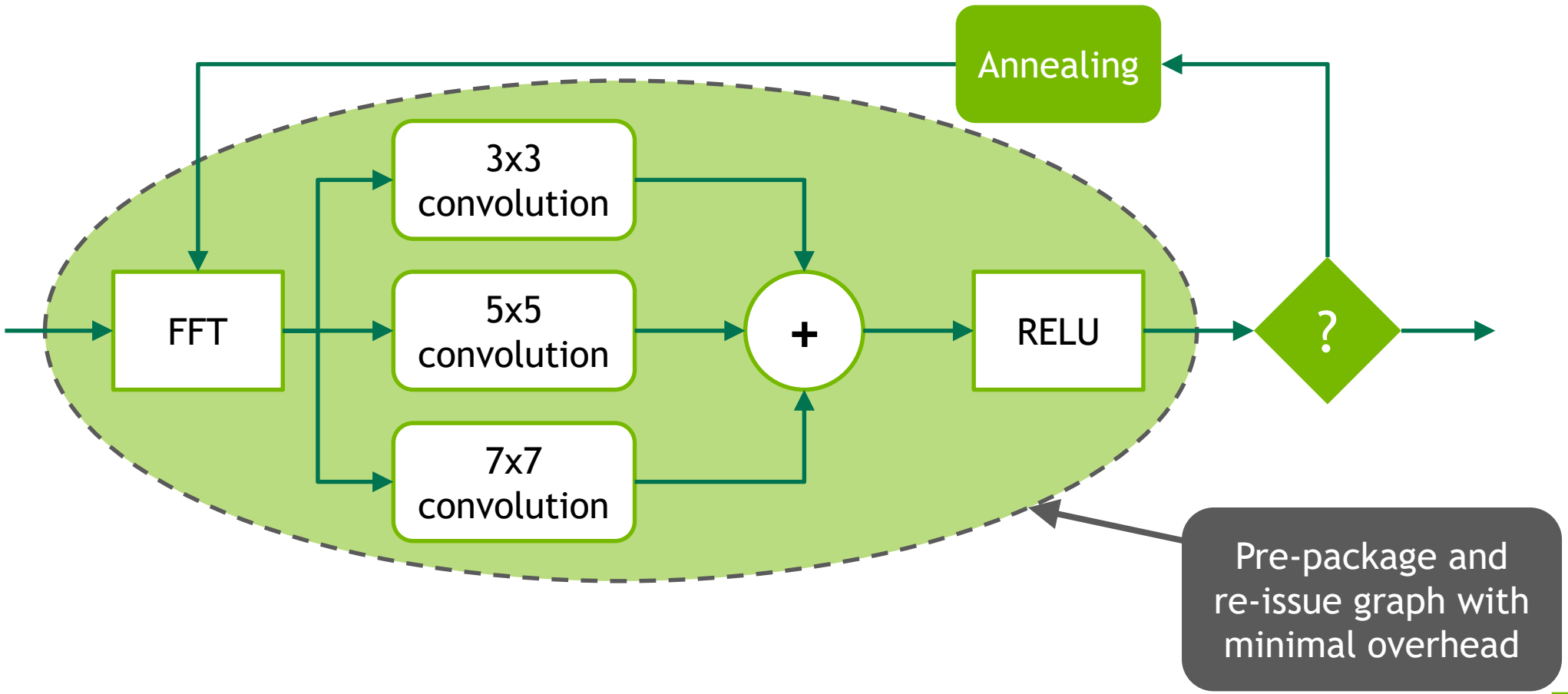
DYNAMIC WORKFLOW ON GPU



KERNEL MERGING



SUB-GRAPH EXECUTION



DYNAMIC RUNTIME PROVISIONING