HiHAT - Q&A

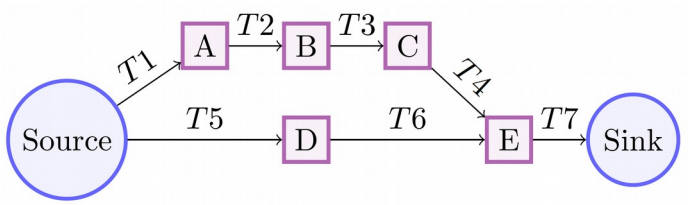
Alexandre Bardakoff, Timothy Blattner, Walid Keyrouz
NIST | ITL | SSD | ISG

National Institute of
Standards and Technology
U.S. Department of Commerce

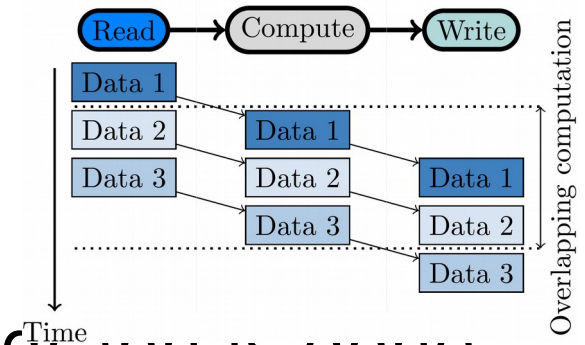
INFORMATION
TECHNOLOGY
LABORATORY

Recap

Data flow graph with T-types and connected nodes



Data pipelining graph

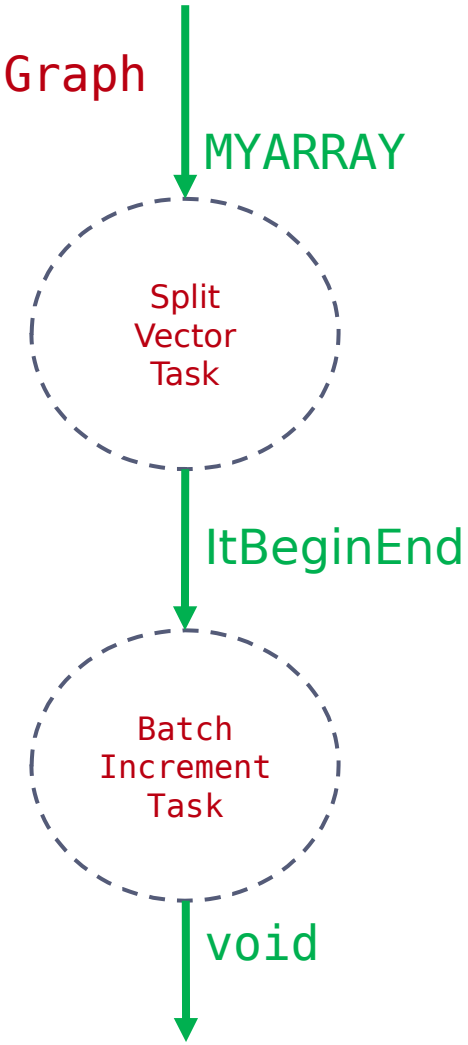


- Data flow
- Data pipelining
- Persistent kernel: task loops forever until the graph is done
- One input data is enough to fire/invoke a task

- Coarse grain parallelism
- Asynchronous execution
- Compute kernels don't do state maintenance

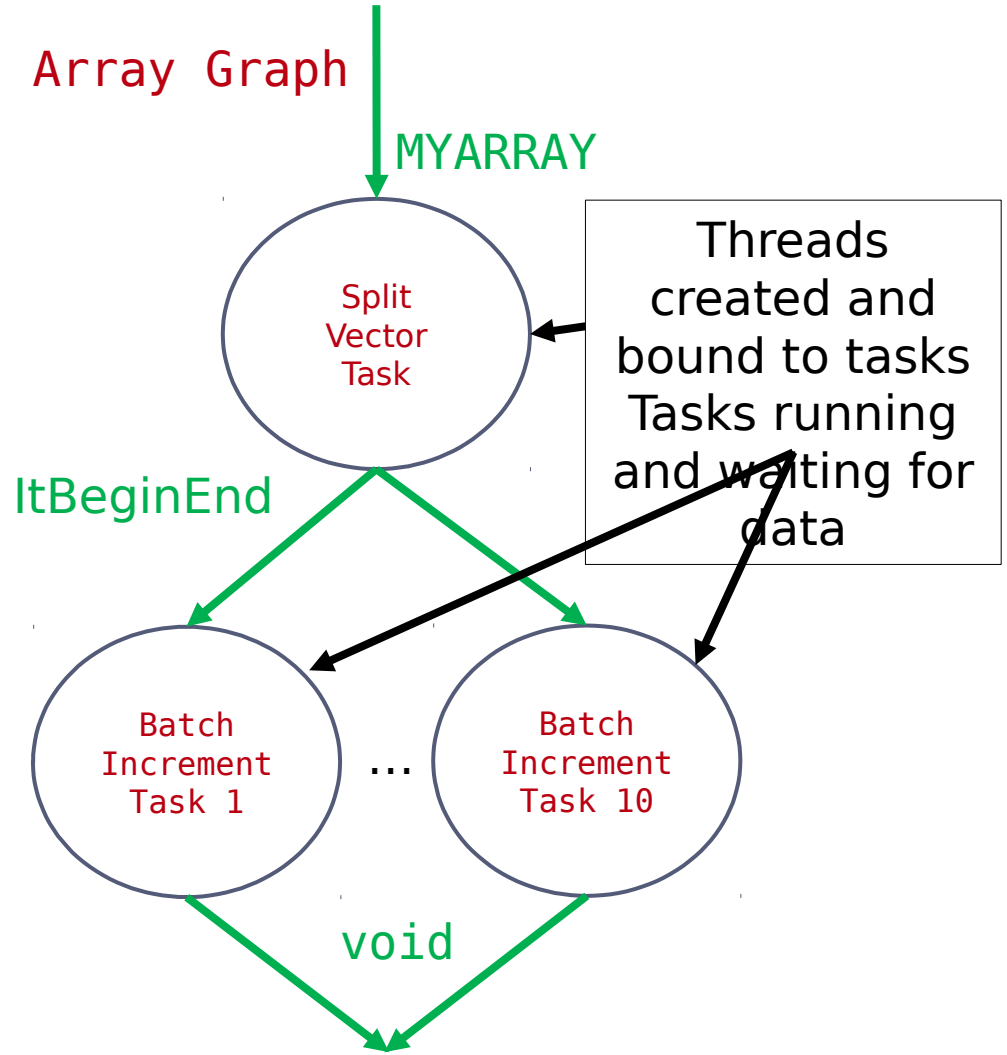
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask = std::make_shared<BatchIncrement>
(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot", hh::ColorScheme::EXECUTION,
hh::StructureOptions::ALL);
}
```



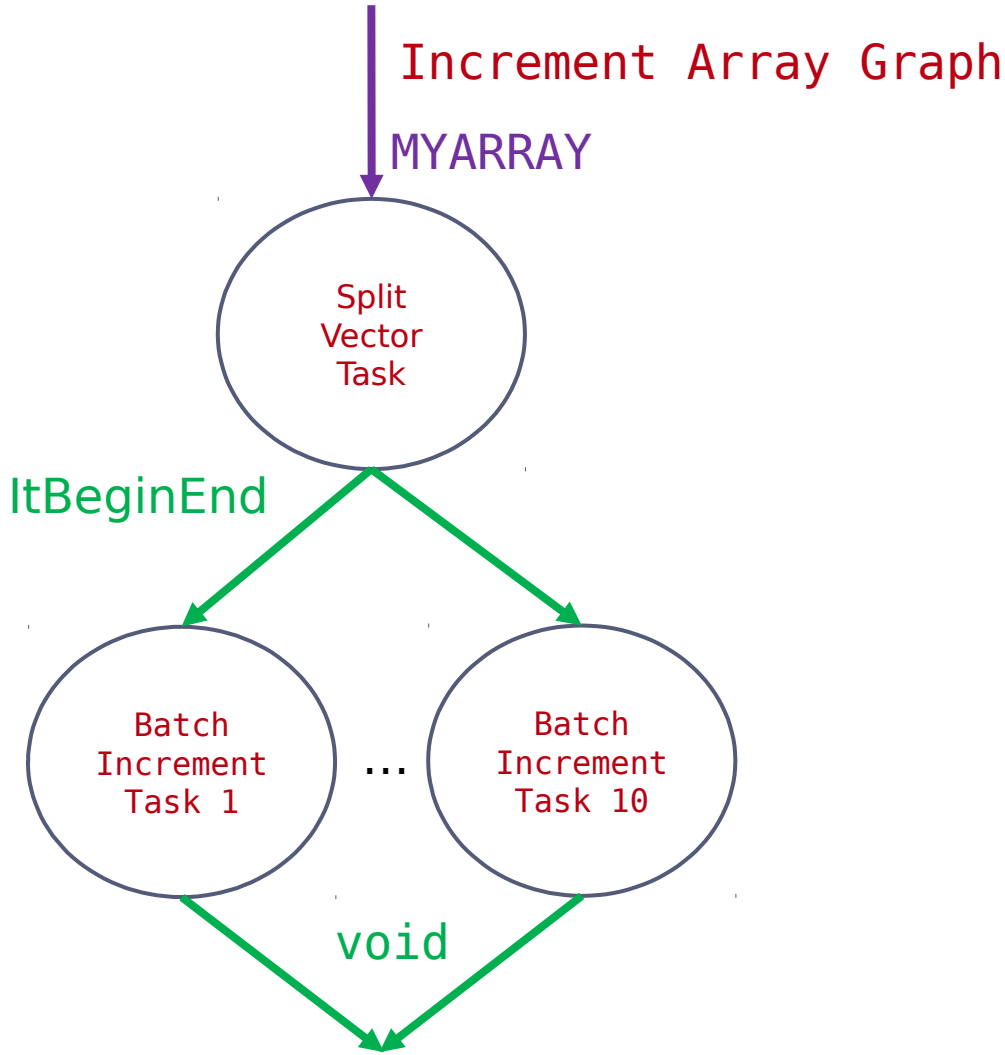
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



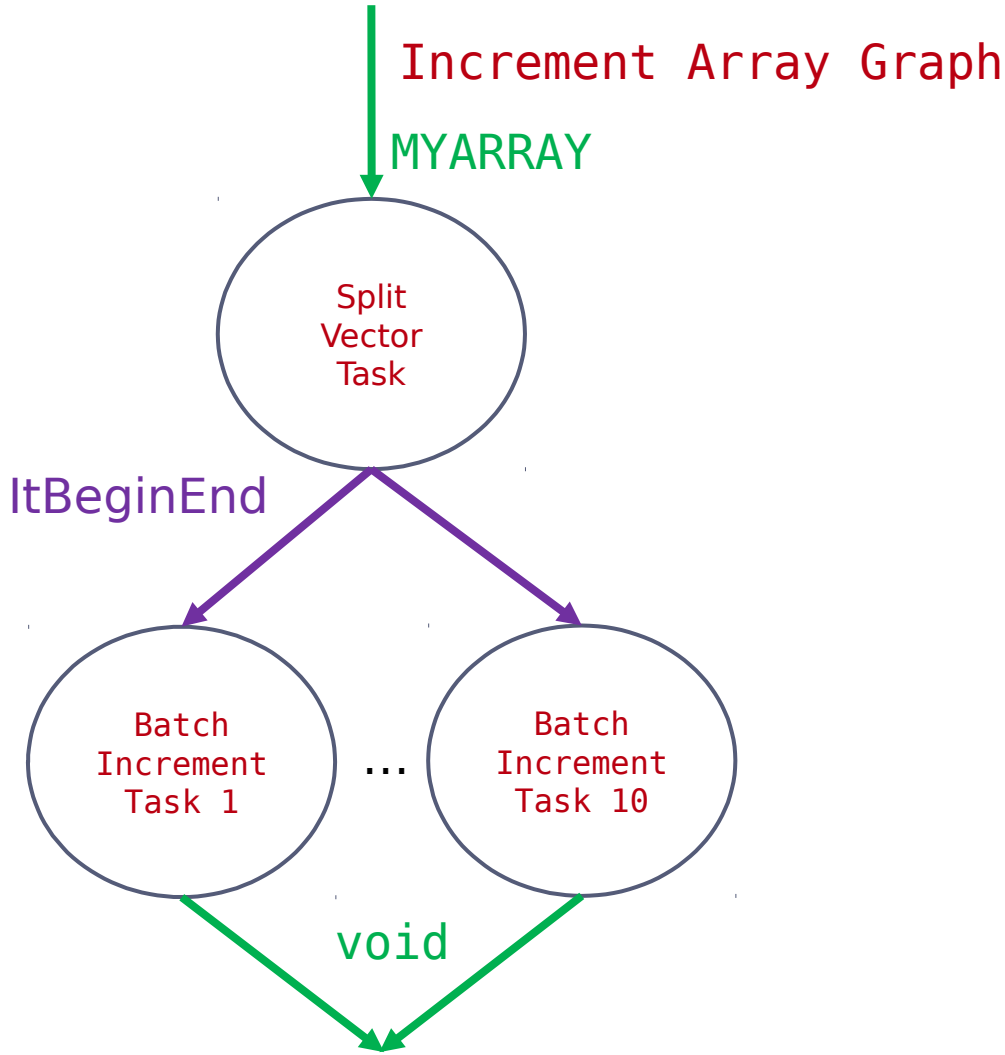
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



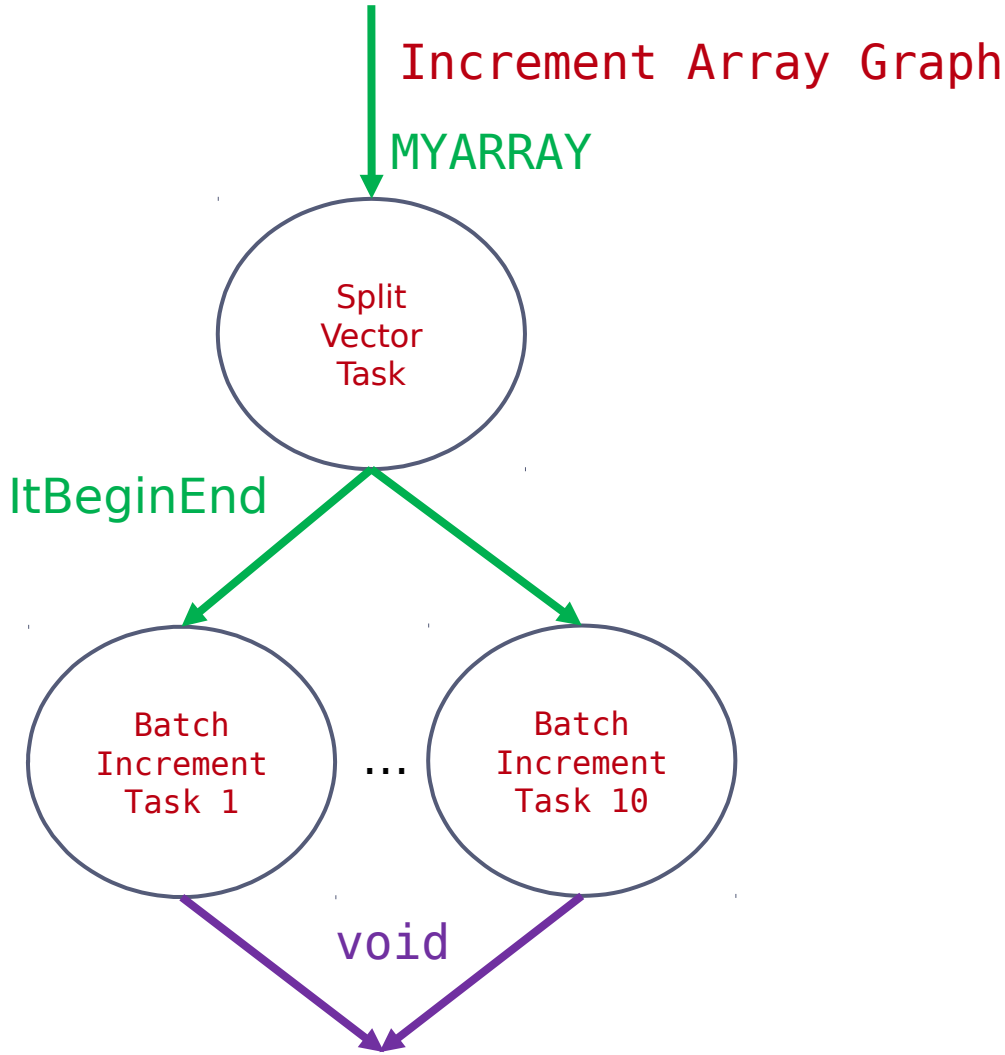
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



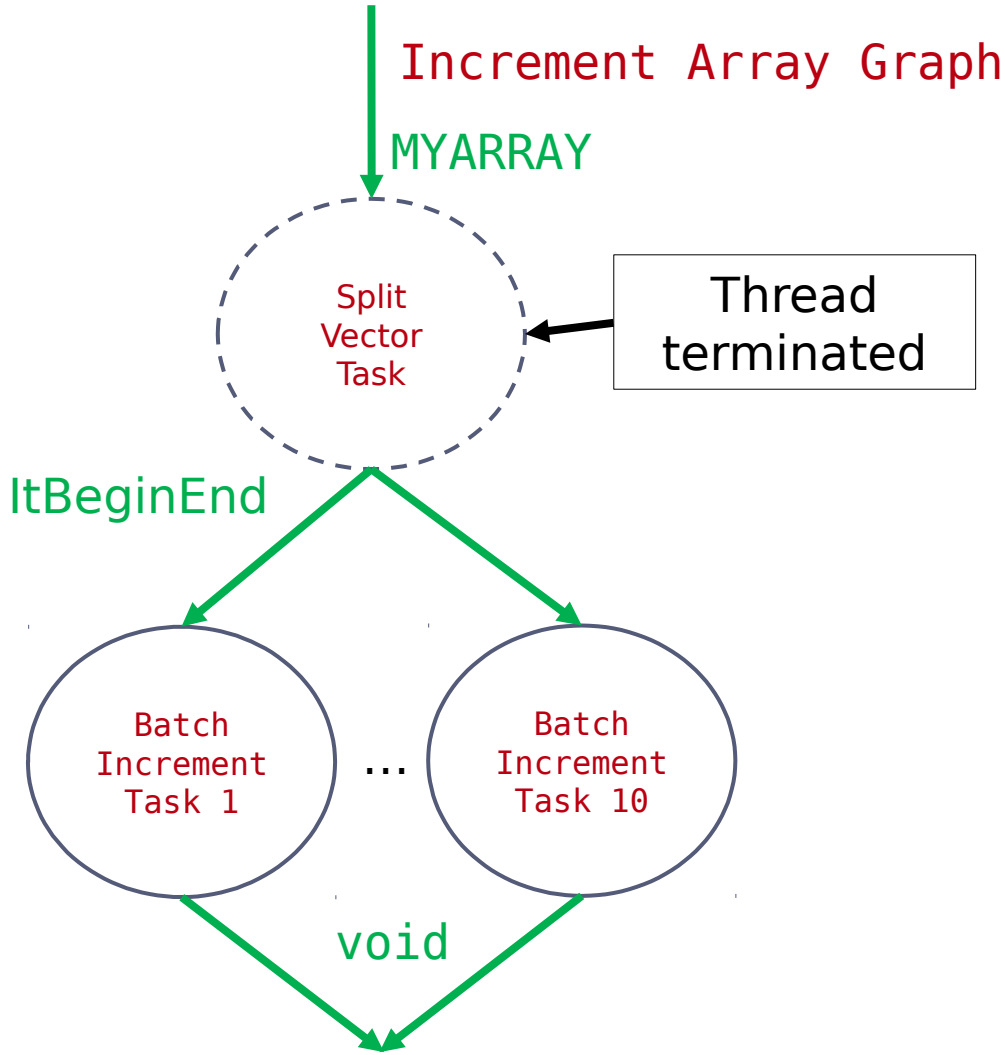
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



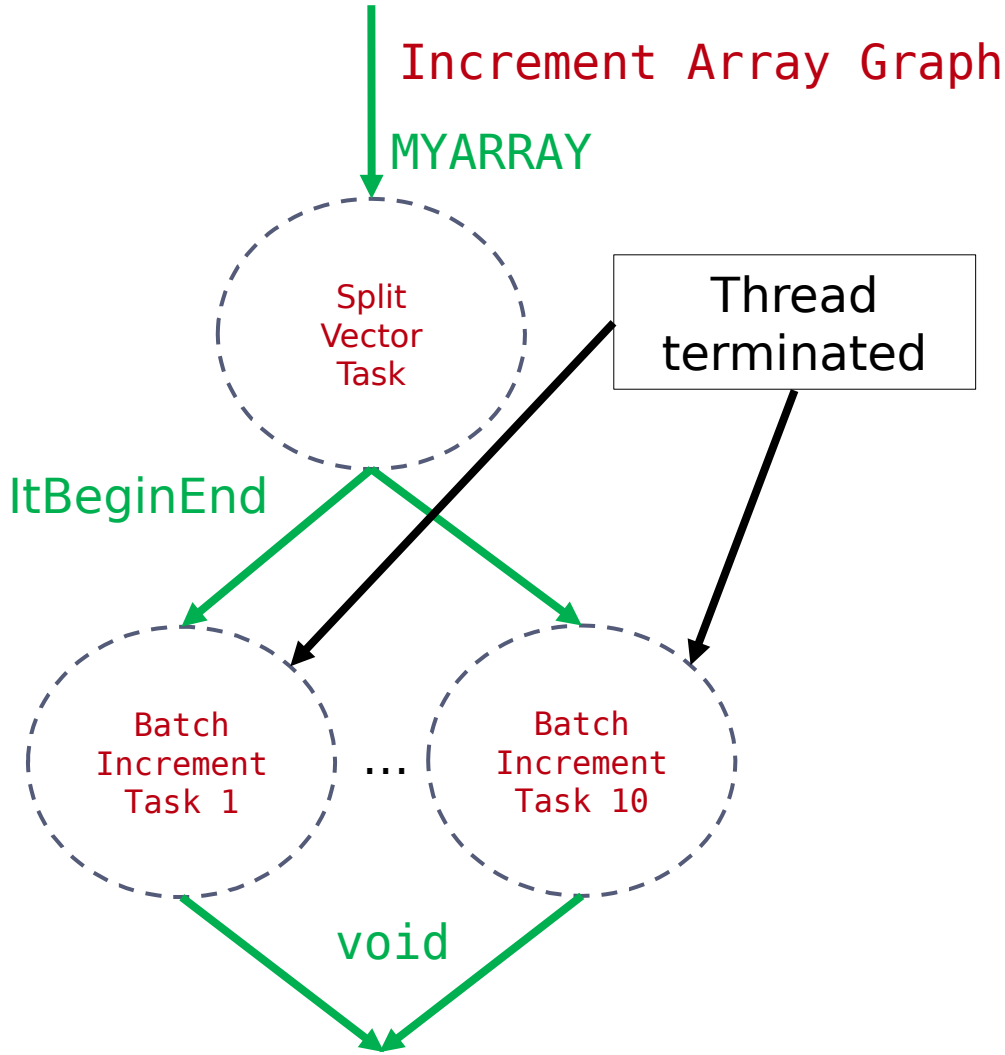
Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



Explicit graphs / Hedgehog data flow

```
int main() {
  auto  myArray = std::make_shared<MYARRAY>();
  auto graph = std::make_shared<hh::Graph<void,
MYARRAY>>("Increment Array Graph");
  auto splitVectorTask = std::make_shared<SplitVector>(1000);
  auto batchIncrementTask =
std::make_shared<BatchIncrement>(100, 10);
  graph->input(splitVectorTask);
  graph->addEdge(splitVectorTask, batchIncrementTask);
  graph->output(batchIncrementTask);
  graph->executeGraph();
  graph->pushData(myArray);
  graph->finishPushingData();
  graph->waitForTermination();
  graph->createDotFile("Test.dot",
hh::ColorScheme::EXECUTION, hh::StructureOptions::ALL);
}
```



User Specification of Types for Nodes

Q: Why is identification of input and output nodes by user necessary? Because compile time only?

- Every node is specialized by its input & output template types
  - Operates on its input types
    - execute(Input_1), execute(Input_2), …, execute(Input_N)
  - Produces its output type
    - addResult(Output)
- To instantiate a node, these templates must be defined
- This allow us to check with traits if the nodes are compatible with their use

Push vs. Pull Model & Work Creation

Q: Push doesn't preclude a pull model. How do tasks create data and generate work?

- Tasks inherit from a pure abstract class, Execute

- Specialized tasks overload from Execute class:

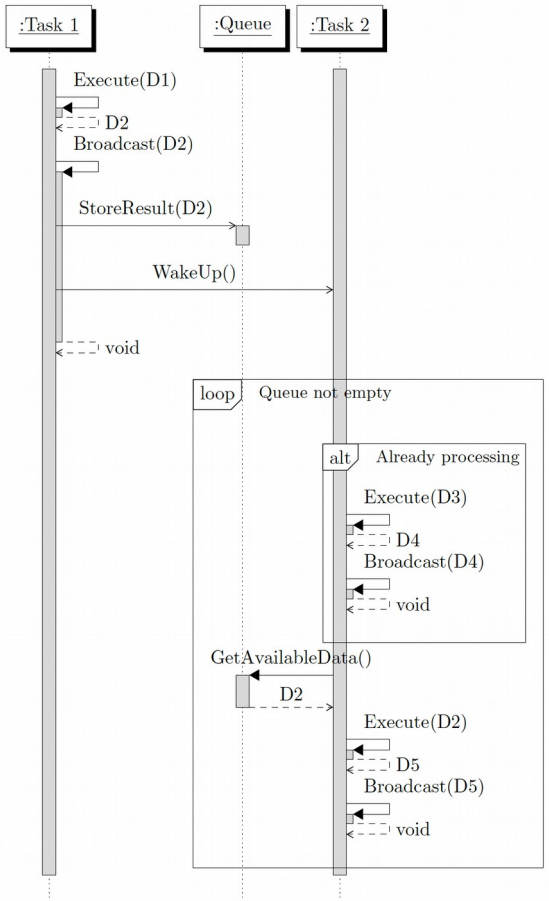
```
virtual void execute(std::shared_ptr<Input>) = 0;
```

- This execute method is called by the task's thread when the corresponding input (of type Input) is available.

- Tasks have a method:

```
void addResult(std::shared_ptr<TaskOutput> output)
```

  - Explicitly called by developer in execute's implementation
  - Task's thread adds the "output" data to the queue(s) between the task's node and all successor node(s)
    - Signal is called to awaken one thread from the successor's threads



UML sequence diagram task I/O

Multiple output data

Q: Can a task have many out edges that get triggered prior to the completion of the whole task? For example, could split vector task have let a batch increment task (that no longer subdivides) start working as soon as 1/10th of its work, i.e. the first

- Asynchronous nature of Hedgehog: as soon as data is available, it begins executing
- When a task calls “addResult(data)” from its “execute” function:
  1. *data* is inserted into the successor task’s input queue
  2. Signal is called to awaken a thread from the successor’s task group
  3. The successor task’s thread awakens and dequeues *data* and invokes its corresponding “execute”

- If a task has multiple outputs, then the data is inserted into multiple queues and signals each of its successor tasks
- A task in “execute” can call “addResult” multiple times to add multiple data items to out queues/edges
- A task starts when an input is available. On waking up, it examines all its input queues for available data and invokes the corresponding “execute” method

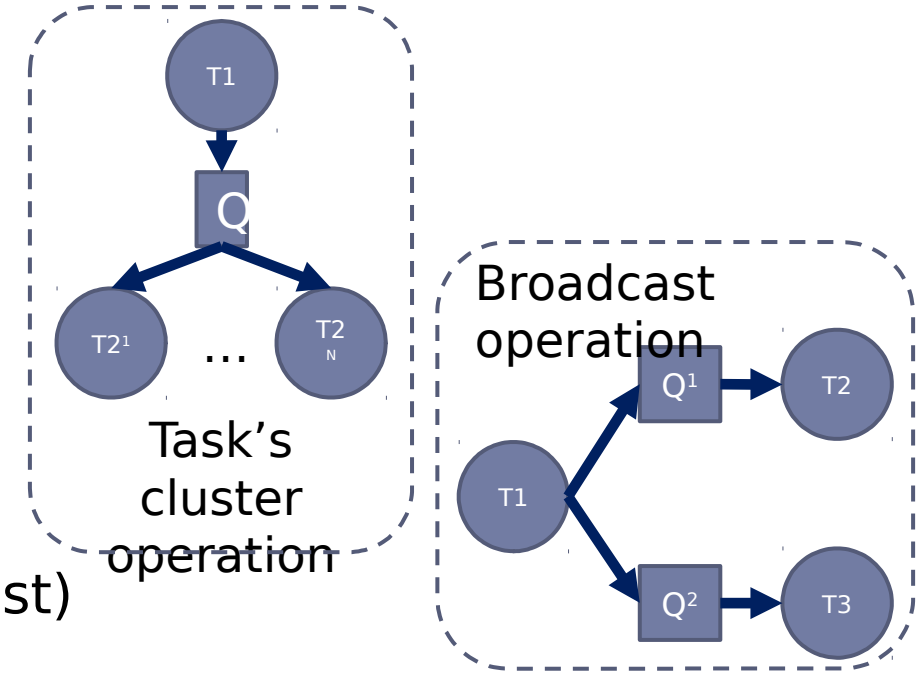
Data movement

Q: Could pushData have been simply a data movement node?

- Not sure about what “data movement node” is
- The data are always wrapped in a shared_ptr
- So what is copied from a node to another is the shared_ptr

- When a task is duplicated, i.e.,
  associated to multiple threads,
  all duplicated tasks are linked to the same queue

- If two tasks are linked to the same task,
  there are 2 queues,
  and the shared_ptr is put in each queue (broadcast)



First block time / Average block time

Q: I missed the talk, but I'd be interested to know where the gap comes from between "first block time" and "average block time". Is the first exceptional, or are all blocks slightly different times (so last block is faster than average)? Is this an implementation detail or an effect of the runtime?

- The first is exceptional because it relates to how fast the next computation can begin.
- Our approach relates to the underlying data pipelining approach. As soon as data is available, the next computation can begin.
- Average block time in our experiments relates to the production rate of blocks after the first one.
- This is an effect of the runtime.
- No guarantees on the rate.

Termination Action/Condition

Q: finishPushingData tells it to not expect any more data beyond what's already been received, so that worker threads can quit. Relevant for persistent kernels.

- Exactly
- finishPushingData sends *terminate* to the graphs and will terminate all nodes in a breadth-first way
- By default, when *terminate* is sent, the graph waits for all inputs to be consumed before terminating
  - Termination of a task can be customized by overloading the "canTerminate" function
    - Can deal with cycles
- Signal sent to task group to check for termination, which cascades
  - Only sends terminate to successor node if all threads in a group have terminated

Dataflow representation

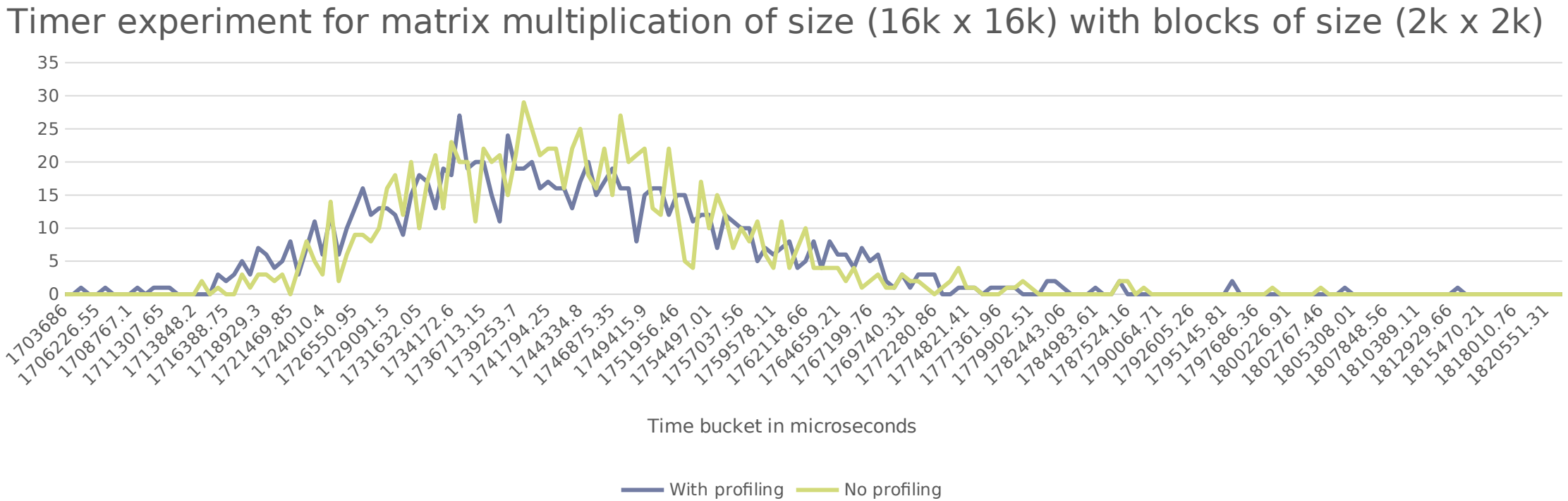
Q: What are the multiple conditions that could trigger dynamic execution, e.g. address (e.g., specific dependence) or generic type? How does a postdominator act on a stimulus from a then or else clause of a preceding (runtime) conditional?

- Hedgehog is based on data flow
- Hedgehog is relying on data pipelining

Feedback cost

Q: How is feedback “costless?”

- We gather information only at node level.
  - Distributions with & w/o statistically indistinguishable.



Group of nodes

Q: What are group nodes? Do they relate to hierarchy?

- Task
  - Instance is cloned forming a group of tasks
    - Number of clones specified by task's constructor
    - Each task in the group is attached to its own thread
- A graph is a node
  - Connected to other tasks/graphs (becomes marked as inner-graph)
  - Outer graph is used by the main thread
- Execution pipeline is a task
  - Owns a graph, that is duplicated
  - Each duplicate graph is attached to a device (specified at Execution pipeline construction)
  - Distribute data with execute function
    - Pipeline Id and device Id denotes which graph to send too

Graph and GPU binding

Q: “Bind a graph to a GPU”. Does this imply graphs do not span GPUs? Do they have external incoming/outgoing edges in order to synchronize with graphs on other GPUs?

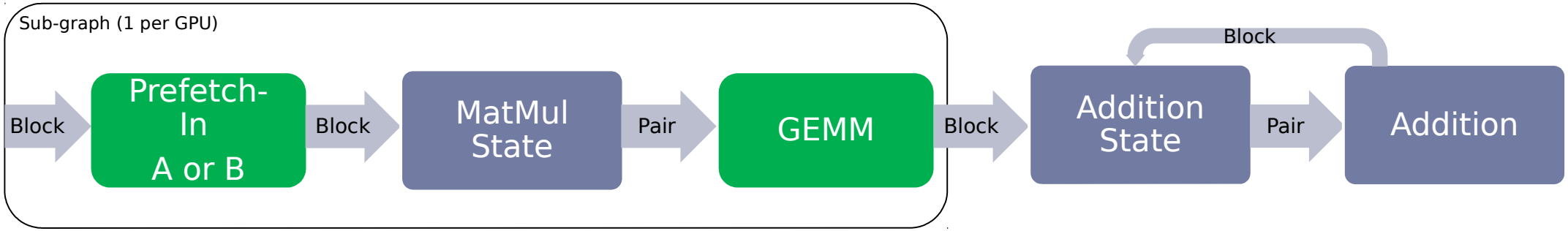
- Our idea 1 Graph = 1 GPU
- If same algorithm on n GPUs Ⓟ Graph duplication Ⓟ Execution pipeline
- Execution pipeline will:
  - Duplicate a graph and associate the graph to a GPU
  - Execute for the execution pipeline can call addResult to send data to the right graph and its corresponding GPU
- If data need to be shared amongst different GPUs during the same algorithm
  - Device Id can be stored in data, depending on boundaries for the algorithm, can be used to initiate copies between GPU memory addresses
    - CUDA tasks can also automatically enable peer access when available

Shutdown and GPU

Q: “Shutdown virtual method to break cycles”. This has implications on the underlying hardware which may introduce inefficiencies (for a GPU, this would be the case as upcoming work would not be able to be pre-fetched). A shutdown conditional node would solve this (i.e. prefetch disabled only for the successor to the shutdown node) but limits the locations where shutdown can occur. There are many other approaches: what does Hedgehog have in mind here?

- Hedgehog uses persistent kernels
  - Thread is bound to a task that gets invoked multiple times based on data flow
  - If the thread terminates and that task will no longer invoke
- Next slide for example from matrix multiplication

HH-GEMM CUDA Graph



Threads	2 (1 stream per thread)	1	5 (1 stream per thread)	1	N
Functionality	HH Get $Mem_{A \mid B}$ Prefetch $Mem_{A \mid B}$ CPU ☾ GPU Create $Event_1$	Pair Mem_A and Mem_B (based on MatMul)	HH Get $Mem_{Partial(P)}$ Prefetch Mem_P CPU ☾ GPU Synchronize $Event_1$ cublasSgemm(Mem_P ,Mem_A , Mem_B) Synchronize Stream Recycle $Mem_{A \& B}$ Prefetch Mem_P GPU ☾ CPU Create $Event_2$	Pair Mem_P with C Decrement ttl	Synchronize $Event_2$ C = Mem_P + C …le Mem_P

```
Notes about cycle termination:
ttl = nBlocks^3
// number of times addition is done
// time to live

bool canTerminate() {
  return ttl == 0;
}
```

Asynchronous pre-fetch

Q: My own experiments on transparent multi-GPU execution indicate that performance is very sensitive to data locality. Is there a pinning operation for the data? Do you assume the pre-fetch is persistent? Would data migration be beneficial at all? What is the granularity with which these pre-fetches can happen?

- Hedgehog by itself only provides a pool of available data. The developer decides how data are pre-fetched
- No pinning by the Hedgehog library (up to the user to specify this)
- Peer access can be enabled by the CudaTask
- Structure the dataflow into the graph to maximize locality, no guarantees
- Depends on domain decomposition for granularity, done by user
  - Inner vs outer product of GPU

State management

- **What does the system for managing state look like?**
- **What is per-node and what is system wide?**

- A state Manager:
  - Is a specialized *single-threaded* task
    - Need a state to be constructed
      - State should be light-weight
    - When execute is invoked:
      1. Lock the state
      2. Send the input data to the state
      3. Gather the output data from the state
      4. Unlock the state
      5. Send the output data from the state to the rest of the graph
- No representation of global state
  - State is local, only between a StateManager and its inputs and outputs
    - Task1 Ⓟ StateManager Ⓟ Task2

OMP/MPI

- **Per above, how are triggering conditions specified? Can that be integrated into inter-node comms like MPI via OMP tasking?**

- OMP can be done within a task
  - Haven’t had a need for this yet…
- MPI requires strict ordering for data transfers
  - Hedgehog is fully asynchronous, so very difficult
  - MPI communication can be done outside of the graph, but incurs significant overhead
    - Ideally would like asynchronous MPI
      - Multi-threaded support
      - Does not rely on send/receive pairs (send whenever data is ready)

Memory management

- **Can alloc/free be visible both as a node in a graph and as a code operation inside a graph?**
- **Are special interfaces needed to make alloc/free visible to this dep system, especially for alloc/free in code inside an execution action?**
- **Is the same dependence system used to manage memory availability as for execution and data deps?**
- **Can in or out arcs regarding memory availability occur from/to the middle of an execution action?**
- **Is the totally memory accessible under admin control?**

- Memory manager is a tool that the tasks use
- A node is connected to a memory manager
  - Can have multiple independent memory managers for multiple tasks
- A memory manager is attached to a task, so the data is acquired in the “execute” method
- If a memory pool is empty, the task that uses that memory pool will wait, other tasks can operate as usual as long as data is available
  - When memory is finished being processed it should be sent back from another task
- Memory manager will deal with alloc/free, and Node will interact with the memory manager
- There are no special interfaces as the alloc/free are encapsulated into memory manager
- No admin control

Data as a first-class citizen

- **In what ways is data treated as a first-class citizen?**

- Hedgehog does asynchronous data flow
  - Rely on data pipelining
  - Schedule is based on the flow of data
- Data is encapsulated into shared_ptr
  - No deep copies

Headers only

- **HiHAT's dispatch is headers only for perf reasons**

- Hedgehog is header only because of:
  - Templates
  - Perf reasons

Compile time focus

- **Burdens the user with identifying input and output nodes?**

- Explicitness of nodes and graphs interfaces, what they accept and produce
  - Clear model
  - Helps collaboration
  - Composable

- Correctness check at compile time

Modern C++

- **SFINAE – can be hard to interpret compile-time error messages**
- **HiHAT didn’t take a templated approach since it used a C ABI and wasn’t all header only.**
- **Others (Tal Ben-Nun did, and Mathias Noack talked about it) did a shim with C++ templates**
- **Could have used traits, since just used at compile time**

- True about SFINAE
  - We just use it in the memory manager
  - Future usage of C++ Concepts with C++20 standards (little mistake on last time report)

- Traits can help having meaningful message with the usage of std::static_assert

hedgehog/api/../behavior/io/../../core/node/core_graph.h:289:5: error: static_assert failed due to requirement 'traits::Contains_v<MatrixBlockData<int, 'a', Order::Row>, std::__1::tuple<MatrixData<int, 'b', Order::Row> > >' "The given Receiver cannot be linked to this Sender"