HiHAT - Q&A

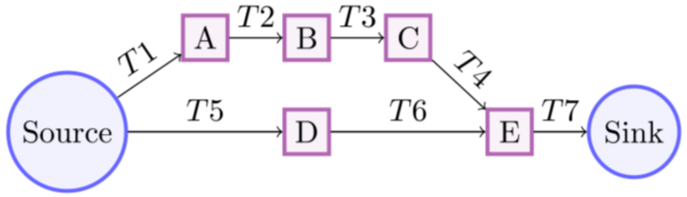
Alexandre Bardakoff, Timothy Blattner, Walid Keyrouz
NIST | ITL | SSD | ISG

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

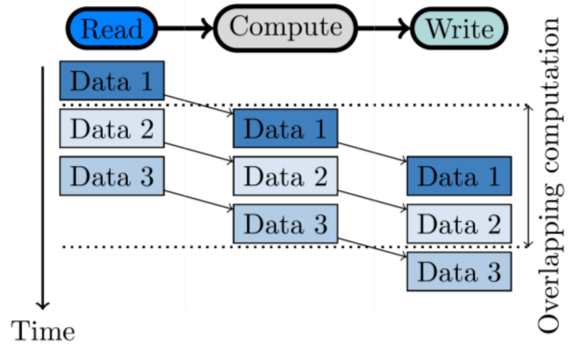
INFORMATION TECHNOLOGY LABORATORY

Recaps

Data flow graph



Data pipelining graph



- Data Flow
- Data pipelining
- Persistent kernel: tasks lives forever on a runtime loop until the graph is done
- One input is enough to fire a task
- Coarse grain parallelism

Example matrix multiplication workflow

- This example is not the complete matrix multiplication
  - It shows only the CUDA-side of the graph, and will compute only partial results

Dataflow through Hedgehog AbstractState pt. 1

```
class MatrixMulState : public hh::AbstractState<
      std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>, std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>,   // output type
       UnifiedMatrixBlockData<float, 'a’>, UnifiedMatrixBlockData<float, 'b’>>  // input types
{
 private:
  size_t gridHeightLeft_ = 0,  gridSharedDimension_ = 0,  gridWidthRight_ = 0;
  std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'a’>>> gridMatrixA_ = {};
  std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'b’>> gridMatrixB_ = {};
  std::vector<size_t> ttlA_ = {}, ttlB_ = {};

 public:
 MatrixMulState(size_t gridHeightLeft, size_t gridSharedDimension, size_t gridWidthRight) : gridHeightLeft_(gridHeightLeft),
gridSharedDimension_(gridSharedDimension), gridWidthRight_(gridWidthRight) {
   gridMatrixA_ = std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>> (gridHeightLeft_ * gridSharedDimension_, nullptr);
   gridMatrixB_ = std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>> (gridWidthRight_ * gridSharedDimension_, nullptr);

   ttlA_ = std::vector<size_t>(gridHeightLeft_ * gridSharedDimension_, gridWidthRight_);
   ttlB_ = std::vector<size_t>(gridWidthRight_ * gridSharedDimension_, gridHeightLeft_);
  }
/* Execute on data (next slide) */  …
 private:
  std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>> matrixA(size_t i, size_t j) {
                 /*Decrement ttl and get matrix from vector*/  …  return = gridMatrixA_[i * gridSharedDimension_ + j];  }
  std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>> matrixB(size_t i, size_t j) {
                 /* Dececrement ttl and get matrix from vector */ …  return = gridMatrixA_[i * gridSharedDimension_ + j]; }
  void matrixA(std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>> blockA) {
                 gridMatrixA_[blockA->rowIdx() * gridSharedDimension_ + blockA->colIdx()] = blockA; }
  void matrixB(std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>> blockB) {
                 gridMatrixB_[blockB->rowIdx() * gridWidthRight_ + blockB->colIdx()] = blockB;  }
```

Dataflow through Hedgehog AbstractState pt. 2

```
void execute(std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>> ptr) override {
  matrixA(ptr); // store ptr into A vector
  for (size_t jB = 0; jB < gridWidthRight_; ++jB) {
    if (auto bB = matrixB(ptr->colIdx(), jB)) {
      // update ttlA because it is used
      auto res = std::make_shared<
                 std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a’>>, std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>>();
      res->first = ptr;
      res->second = bB;
      this->push(res);
    }
  }
}

void execute(std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>> ptr) override {
  matrixB(ptr); // store ptr into B vector
  for (size_t iA = 0; iA < gridHeightLeft_; ++iA) {
    if (auto bA = matrixA(iA, ptr->rowIdx())) {
      // update ttlB because it is used
      auto res = std::make_shared<
                 std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a’>>, std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>>();
      res->first = bA;
      res->second = ptr;
      this->push(res);
    }
  }
}
```

Dataflow through a Hedgehog Task pt. 1

```
class CudaProductTask : public hh::AbstractCUDATask<UnifiedMatrixBlockData<float, 'p’>,
                                        std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>, std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>> {
 private:
  cublasHandle_t handle_ = {};

 public:
  explicit CudaProductTask(size_t numberThreads = 1) : hh::AbstractCUDATask<UnifiedMatrixBlockData<float, 'p’>,
                                        std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>,
                                        std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>>("CUDA Product Task", numberThreads, false, false) {}


  /* Execute on data (next slide) */  ...
  void initializeCuda() override {
    checkCudaErrors(cublasCreate_v2(&handle_));
    checkCudaErrors(cublasSetStream_v2(handle_, this->stream()));
  }

  void shutdownCuda() override {
    checkCudaErrors(cublasDestroy_v2(handle_));
  }

  std::shared_ptr<CudaProductTask>  copy() override { return std::make_shared<CudaProductTask>(this->numberThreads());  }


};
```

Dataflow through a Hedgehog Task pt. 2

```
 void execute(std::shared_ptr<std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>, std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>> ptr) override {
  float alpha = 1., beta = 0.;
  auto matA = ptr->first;
  auto matB = ptr->second;
  auto res = this->getManagedMemory();

  res->rowIdx(matA->rowIdx()); res->colIdx(matB->colIdx());
  res->blockSizeHeight(matA->blockSizeHeight()); res->blockSizeWidth(matB->blockSizeWidth());
  res->leadingDimension(matA->blockSizeHeight());
  res->ttl(1);

  checkCudaErrors(cudaMemPrefetchAsync(res->blockData(), sizeof(float) * res->blockSizeHeight() * res->blockSizeWidth(), this->deviceId(), this->stream()));

  matA->synchronizeEvent();
  matB->synchronizeEvent();

 checkCudaErrors(cublasSgemm_v2( handle_, CUBLAS_OP_N, CUBLAS_OP_N,
                             matA->blockSizeHeight(), matB->blockSizeWidth(), matA->blockSizeWidth(), &alpha, matA->blockData(), matA->leadingDimension(),
                             matB->blockData(), matB->leadingDimension(), &beta, res->blockData(), res->leadingDimension()));

  checkCudaErrors(cudaStreamSynchronize(this->stream()));

  matA->returnToMemoryManager(); matB->returnToMemoryManager();

  checkCudaErrors(cudaMemPrefetchAsync(res->blockData(), sizeof(float) * res->blockSizeHeight() * res->blockSizeWidth(), cudaCpuDeviceId, this->stream()));
  res->recordEvent(this->stream());
  this->addResult(res);
 }
```

Connecting with Hedgehog Graph

```
auto graph = std::make_shared<hh::Graph<UnifiedMatrixBlockData<float, 'p’>, UnifiedMatrixBlockData<float, 'a'>, UnifiedMatrixBlockData<float, 'b’>>>("GPU Computation Graph");

size_t n = 65536, m = 65536, p = 65536, blockSize = 8192, numberThreadProduct = 5, nBlocks = std::ceil(n / blockSize) + (n % blockSize == 0 ? 0 : 1); // same for mBlocks and pBlocks…

auto copyInATask = std::make_shared<CudaPrefetchInGpu<'a'>>(pBlocks);
auto copyInBTask = std::make_shared<CudaPrefetchInGpu<'b'>>(nBlocks);
auto productTask = std::make_shared<CudaProductTask>(numberThreadProduct);

auto cudaMemoryManagerA =
            std::make_shared<hh::StaticMemoryManager<UnifiedMatrixBlockData<float, 'a'>>>(restrictInputMemory ? nBlocks + 4 : nBlocks * mBlocks);
auto cudaMemoryManagerB =
            std::make_shared<hh::StaticMemoryManager<UnifiedMatrixBlockData<float, 'b'>>>(restrictInputMemory ? pBlocks + 4 : pBlocks * mBlocks);
auto cudaMemoryManagerProduct =
            std::make_shared<hh::StaticMemoryManager<UnifiedMatrixBlockData<float, 'p'>, size_t>>(8, blockSize);

productTask->connectMemoryManager(cudaMemoryManagerProduct);
copyInATask->connectMemoryManager(cudaMemoryManagerA);
copyInBTask->connectMemoryManager(cudaMemoryManagerB);

auto stateInputBlock =
            std::make_shared<MatrixMulState>(nBlocks, mBlocks, pBlocks);
auto stateManagerInputBlock =
            std::make_shared<hh::StateManager<
                        std::pair<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>,  std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>>,
                        UnifiedMatrixBlockData<float, 'a'>, UnifiedMatrixBlockData<float, 'b'>>>("Input State Manager", stateInputBlock);

this->input(copyInATask);
this->input(copyInBTask);
this->addEdge(copyInATask, stateManagerInputBlock);
this->addEdge(copyInBTask, stateManagerInputBlock);
this->addEdge(stateManagerInputBlock, productTask);
this->output(productTask);
```

Producing data for the graph

```
graph->execute();

std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'a'>>> aMatrixData;
std::vector<std::shared_ptr<UnifiedMatrixBlockData<float, 'b'>>> bMatrixData;

if (innerTraversal)
   for (size_t i = 0; i < nBlocks; ++i) { for (size_t j = 0; j < mBlocks; ++j) { aMatrixData.push_back(allocateUnifiedMemory<float, 'a'>(i, j, blockSize, unif, rng)); }}
else
   for (size_t j = 0; j < mBlocks; ++j) { for (size_t i = 0; i < nBlocks; ++i) { aMatrixData.push_back(allocateUnifiedMemory<float, 'a'>(i, j, blockSize, unif, rng)); }}

if (innerTraversal)
   for (size_t j = 0; j < pBlocks; ++j) { for (size_t i = 0; i < mBlocks; ++i) { bMatrixData.push_back(allocateUnifiedMemory<float, 'b'>(i, j, blockSize, unif, rng)); }}
else
   for (size_t i = 0; i < mBlocks; ++i) { for (size_t j = 0; j < pBlocks; ++j) { bMatrixData.push_back(allocateUnifiedMemory<float, 'b'>(i, j, blockSize, unif, rng)); }}

// Push the matrices
for (auto data : aMatrixData)
   graph->pushData(data);

for (auto data : bMatrixData)
  graph->pushData(data);

graph->finishPushingData();
graph->waitForTermination();
```

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
- 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?

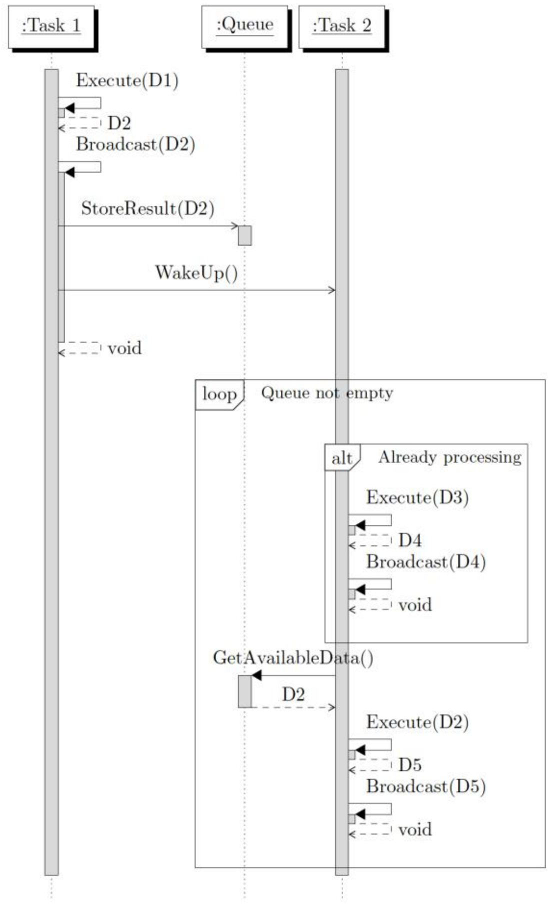
- Tasks inherit from a pure abstract class, Execute
- Specialized tasks overload from Execute class:

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

- This execute method is called by the library 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
  - Adds the "output" data to the queue between the task's node and a successor node



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

- If an "out edge" is activated, all are
- A task in "execute" can call "addResult" 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

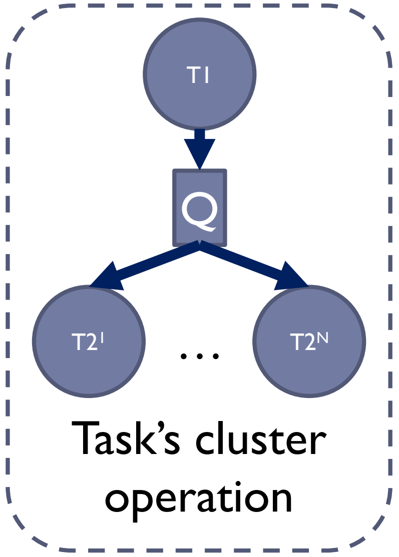
Data movement

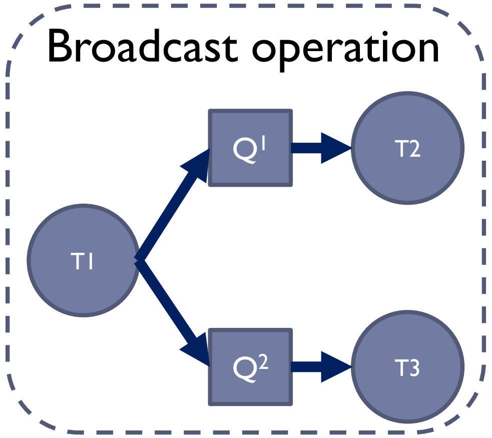
Q: Could pushData have been simply a data movement node?

- 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

Dataflow representation

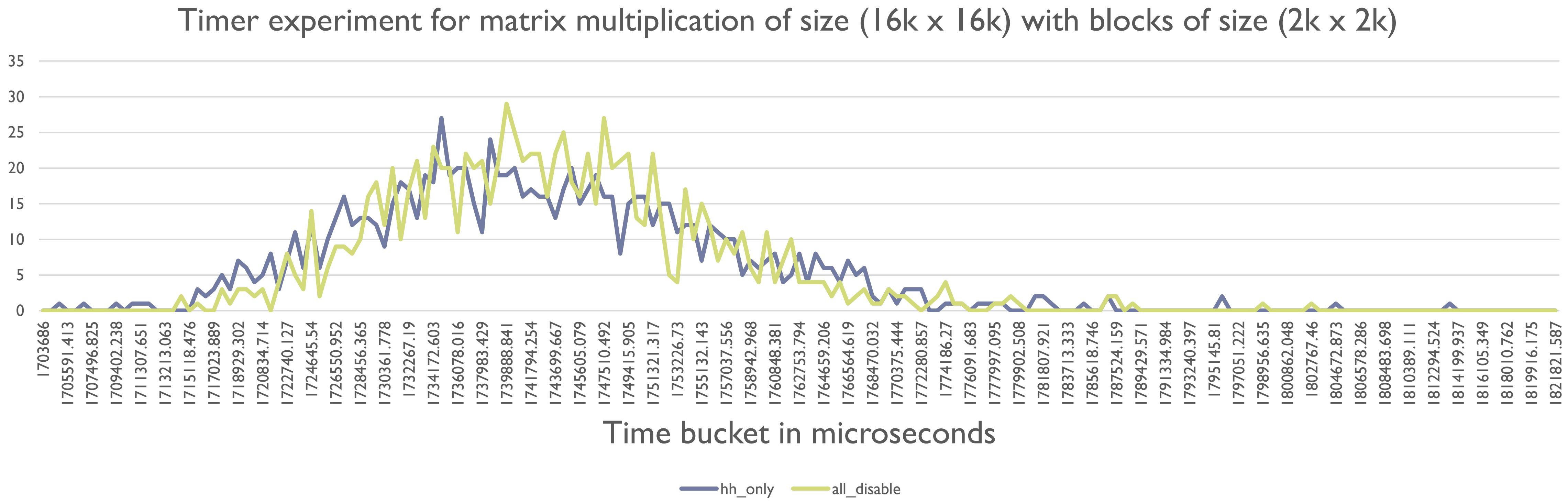
Q: What are the multiple conditions that could trigger dynamic execution, e.g. address (e.g., specific dependence) or generic type? How does a postdominator act on a stimulus from a then or else clause of a preceding (runtime) conditional?

- 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?

- I don’t know what you relate to
- A cluster of task, or a group of task are the set of task plus its clone
- Construct of the graph as a node
- Execution pipeline

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
  - Have a rule to redirect an execution pipeline input to the right graph so the right GPU
- If data need to be shared amongst different GPU during the same algorithm, a state & state manager can be used amongst the graph's copies
  - 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

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?

- Tim any thoughts?

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

- A state Manager:
  - Is a specialized *single-threaded* task
    - That can't be duplicated
    - Need a state to be constructed
  - 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
- We do not represent system state
  - State is local, only between a StateManager and its inputs and outputs
    - Task1 ➔ StateManager ➔ Task2
- We do not have a way to represent the global state of the computation, the state manager is only a graph's node, so will deal only with local information
- Tim ? Walid ➔
- OMP can be done within a task if needed
- 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

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 totally memory accessible under admin control?**
- **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?**

- 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
- 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
- There is no admin control, user controls pool size
- A memory manager is attached to a state manager or a task, so the data is acquired in "execution" 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

Data as a first-class citizen

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

- Data is encapsulated into shared_ptr
  - Avoid not needed real data movement
- We tried to make data management easier for computation on really small computers, or limited-memory devices as GPUs
- Data in Hedgehog controls the scheduling and parallelism of our nodes and graphs
  - Rely on data pipelining

Headers only

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

- Hedgehog is header only because of:
  - The 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
- 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

Backup