Hedgehog: Understandable Scheduler-Free Heterogeneous Asynchronous Multithreaded Data-Flow Graphs

Alexandre Bardakoff

Bruno Bachelet – Timothy Blattner – Walid Keyrouz – Gerson Kroiz – Loïc Yon

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

Université Clermont Auvergne

LIMOS

INFORMATION TECHNOLOGY LABORATORY

Disclaimer

No approval or endorsement of any commercial product by NIST is intended or implied. Certain commercial software, products, and systems are identified in this report to facilitate better understanding. Such identification does not imply recommendations or endorsement by NIST, nor does it imply that the software and products identified are necessarily the best available for the purpose.

Context: High-end heterogeneous node

- High degree of hardware parallelism
  - High core count CPU, 2×64 cores (+ hyperthreading)
  - Multiple accelerators and GPUs, 4 to 8 per node
  - Hardware is widely accessible
- To obtain performance with software:
  - Compose optimized compute kernels and efficient data transfers
  - Express locality of data
  - Overlap data motion and computation
- Given that high-end hardware is widely available, software should:
  - Be approachable by non-HPC experts
  - Have an understandable program and execution model

Problem

How to develop an **understandable parallel** program/algorithm on a **heterogeneous** node?

MODEL

- Explicit
- Understandable

LIBRARY

- High level
  - Expressive
  - Simple
- Targets various hardware platforms
- Feedback mechanism
- Experiment for performance

RESULTS

- Reasonable amount of effort
- Utilizes hardware
  - Keeps hardware busy
- Utilizes existing libraries
- Portable designs for performance

Outline

- Model
- Hedgehog
- Theoretical example
- Experimentations

Model

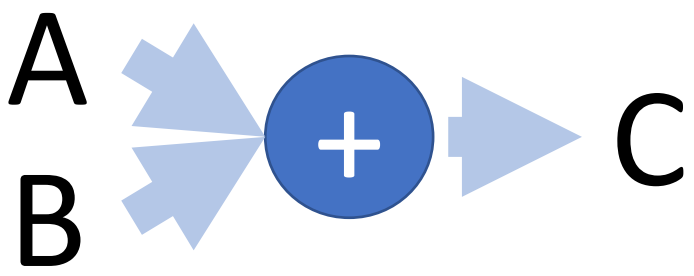
Data flow graph

Data pipelining

Asynchronous Data Flow Graph

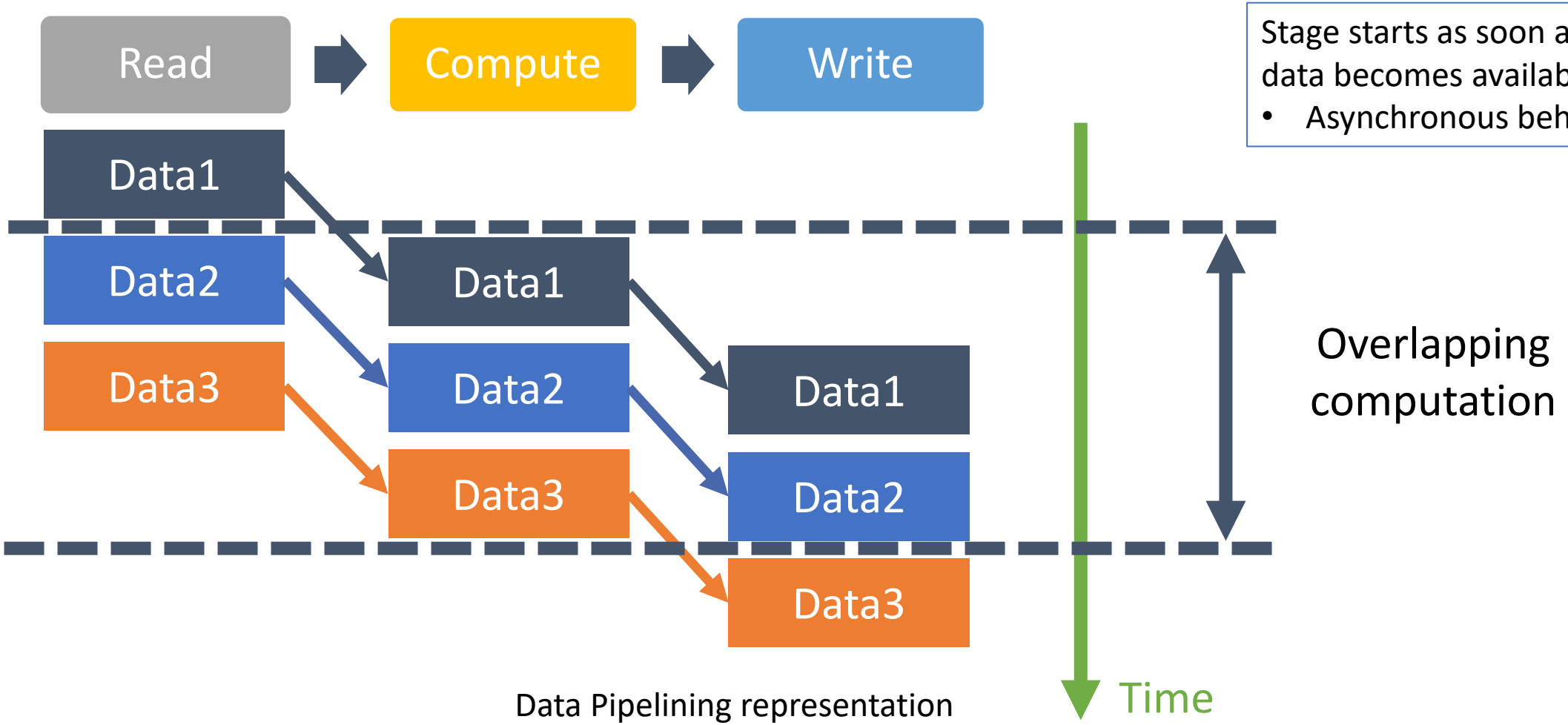
- Program model
  - Directed graph representation
  - 1 entry and 1 exit point (source and sink)
- Components
  - Nodes: computations or state management
  - Edges: directed information flow

Addition algorithm



Data Flow representation

Data Pipelining



Stage starts as soon as data becomes available

- Asynchronous behavior

Data Pipelining representation

Hybrid Task Graph Scheduler - HTGS

- **First implementation of model**
- **Coarse-grained p**arallelism
  - **Pipelined** multi-threaded
  - **Multi-CPU** and m**ulti-GPU**
- **C++ 11** headers-only library
  - **Visual debugging f**eature
  - **Rich** API

- The Hybrid Task Graph Scheduler API
  - https://github.com/usnistgov/HTGS
- A Hybrid Task Graph Scheduler for High Performance Image Processing Workflows, Blattner T. et al., J. Sign. Process. Syst. (2017) 89: 457
  - DOI: 10.1007/s11265-017-1262-6

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

Hedgehog

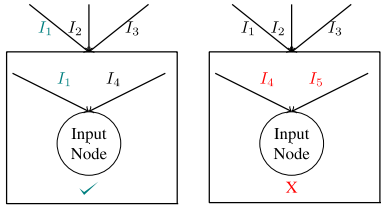
Overview

Methodology

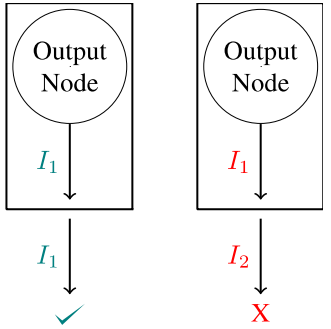
API

Overview

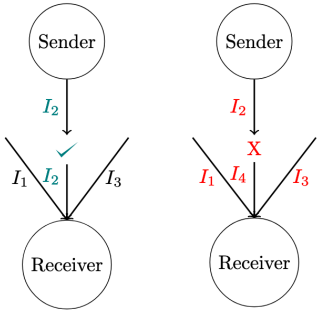
- C++ 17, headers-only library
  - General purpose
  - Open source and available
- Hedgehog design
  - Dataflow graph representation
  - Data pipelining to obtain performance & keep hardware busy
  - Coarse-grained parallelism
  - **No scheduler**
    - Operates based on the presence of data
  - Separation of concerns:
    - Tasks; State; Memory Management
- Metaprogramming for type safety



Input Check

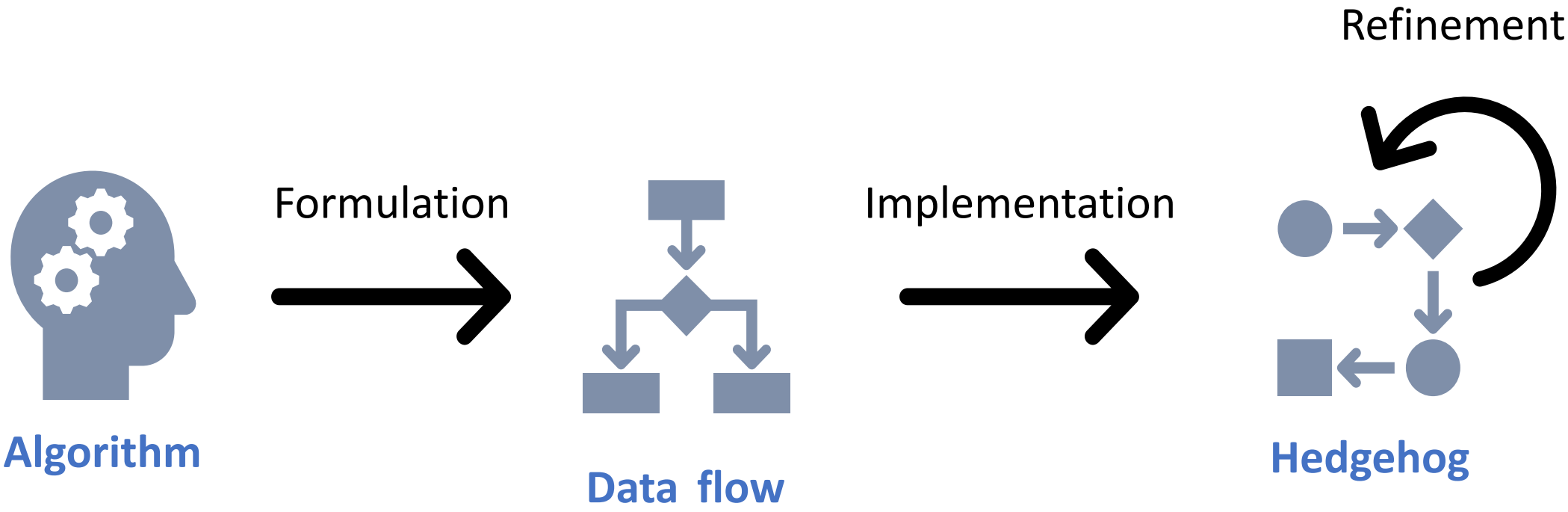


Output Check



Compatibility Check

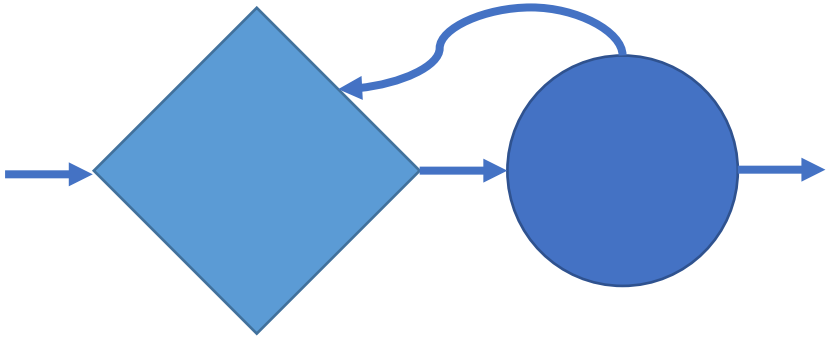
Methodology



Methodology used in Hedgehog

API - Nodes

- **Tasks**
  - Step of an algorithm / **Computation kernels**
  - **Specialized task** for (NVIDIA) GPU computations
  - **Multithreaded**
- State manager (single-threaded)
  - **Local** computation's **state** management
  - State can be shared between different managers in the graph
- Multiple Inputs - Single Output
- *canTerminate*, virtual method, to break cycles



Example of connected nodes
Diamond: State Manager
Circle: Computational Task
Edges: Flow of data

API - Memory Manager

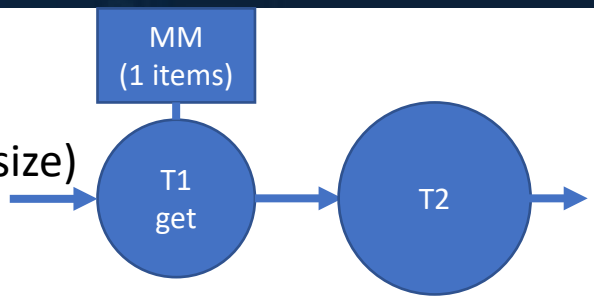
- Throttles memory usage
- Links to a task or state
- Pool of available pieces of data

- Static
  - Creates *n* objects calling a specific constructor
  - Ensures constructor signature

- Dynamic
  - Creates *n* objects calling default constructor

- Mechanism to recycle memory / objects

Memory manager usage
Circle: Computational Task
Rectangle: Memory manager (pool size)
Square: Managed data
Edges: Flow of data

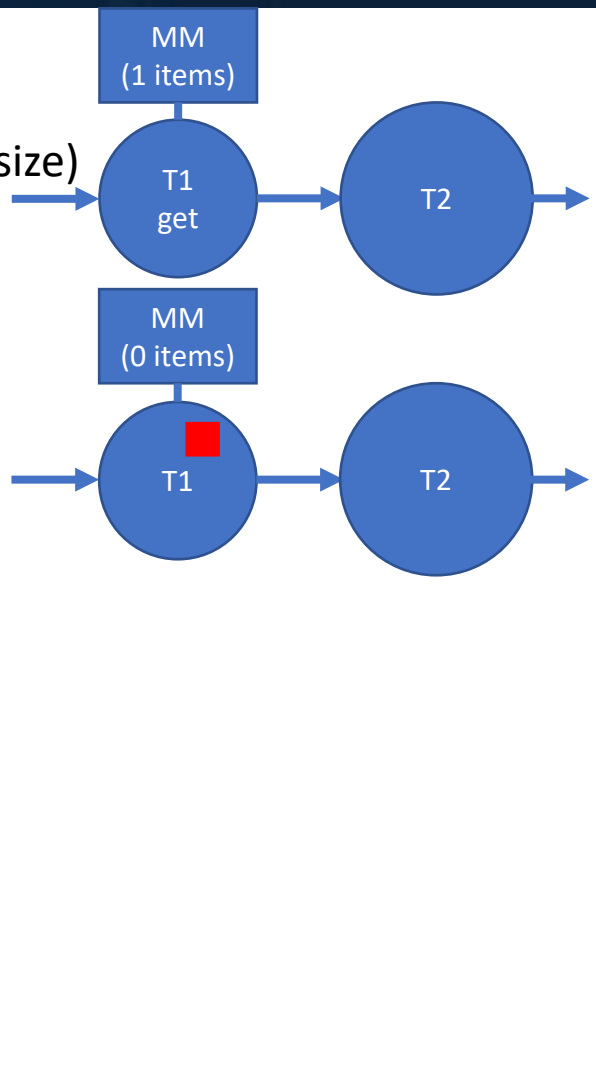


Time

API - Memory Manager

- Throttles memory usage
- Links to a task or state
- Pool of available pieces of data

Memory manager usage
Circle: Computational Task
Rectangle: Memory manager (pool size)
Square: Managed data
Edges: Flow of data



- Static
  - Creates *n* objects calling a specific constructor
  - Ensures constructor signature
- Dynamic
  - Creates *n* objects calling default constructor
- Mechanism to recycle memory / objects

API - Memory Manager

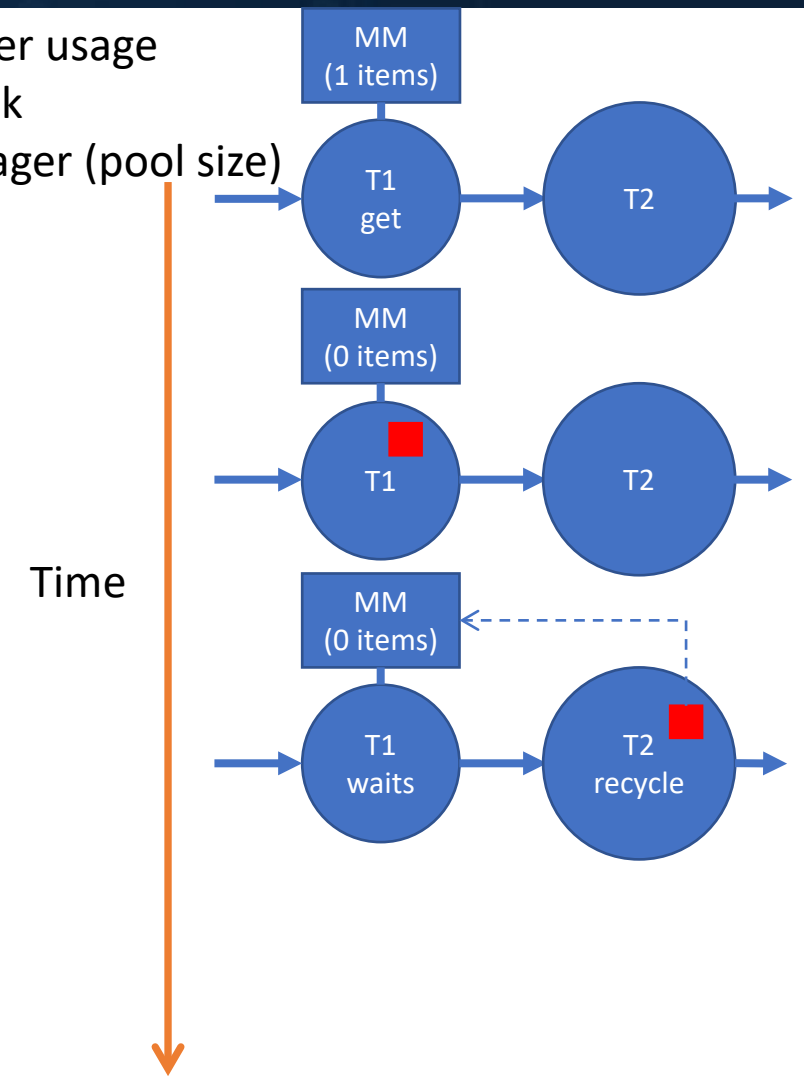
- Throttles memory usage
- Links to a task or state
- Pool of available pieces of data

- Static
  - Creates *n* objects calling a specific constructor
  - Ensures constructor signature

- Dynamic
  - Creates *n* objects calling default constructor

- Mechanism to recycle memory / objects

Memory manager usage
Circle: Computational Task
Rectangle: Memory manager (pool size)
Square: Managed data
Edges: Flow of data



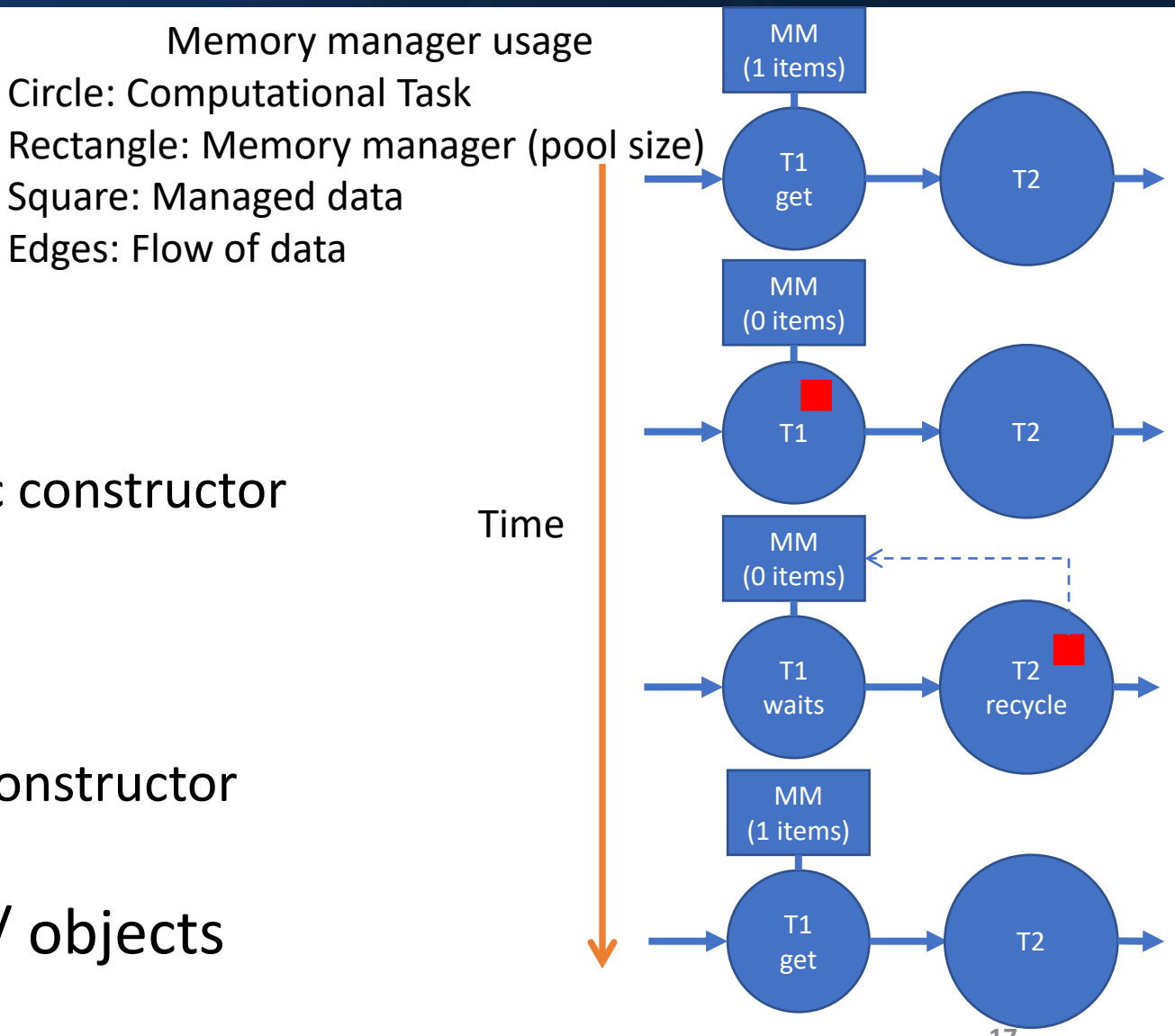
API - Memory Manager

- Throttles memory usage
- Links to a task or state
- Pool of available pieces of data

- Static
  - Creates *n* objects calling a specific constructor
  - Ensures constructor signature

- Dynamic
  - Creates *n* objects calling default constructor

- Mechanism to recycle memory / objects

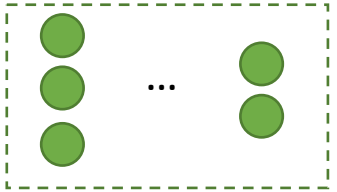


API - Graph

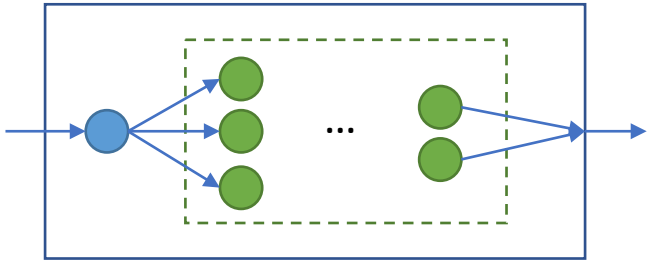
- Graph
  - Algorithm representation
  - Group nodes (tasks, state manager, memory manager)
  - Can be part of another graph
    - Share or compose algorithms
  - Binds a graph to a device
  - Only object used by an end-user

- Execution Pipeline
  - Duplicates graph
    - Data decomposition rules
    - Associates each graph copy to a specific device

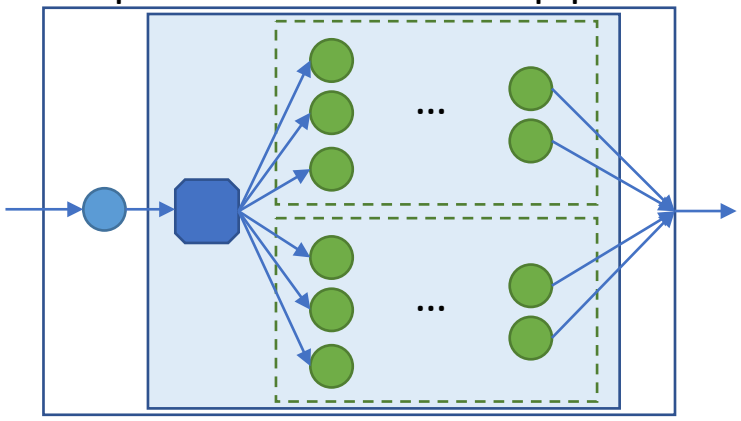
GPU Graph



Graph with an inner graph as output node

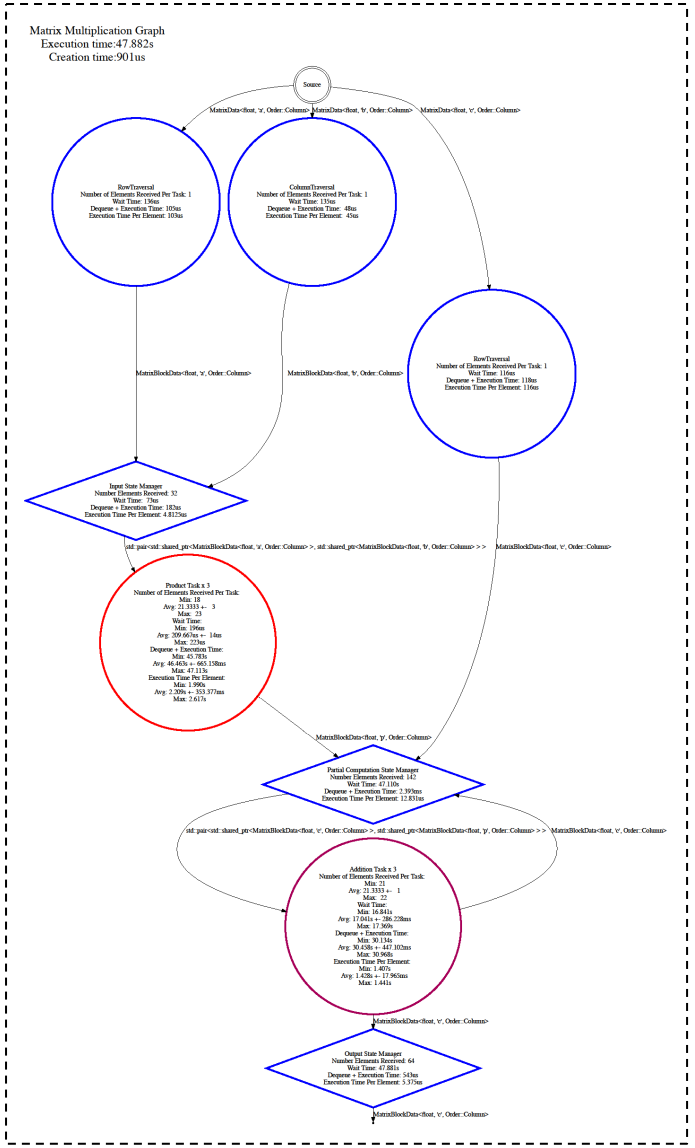


Graph with an execution pipeline as output node



Explicit representation

- Graphical representation
  - Contains profiling
  - Very low overhead (task level)
- Information gathered
  - Graph: execution & creation times
  - Nodes: wait & execution times
- Node colors
  - Based on execution & wait times
- Multiple options (all threads)



Dot representation

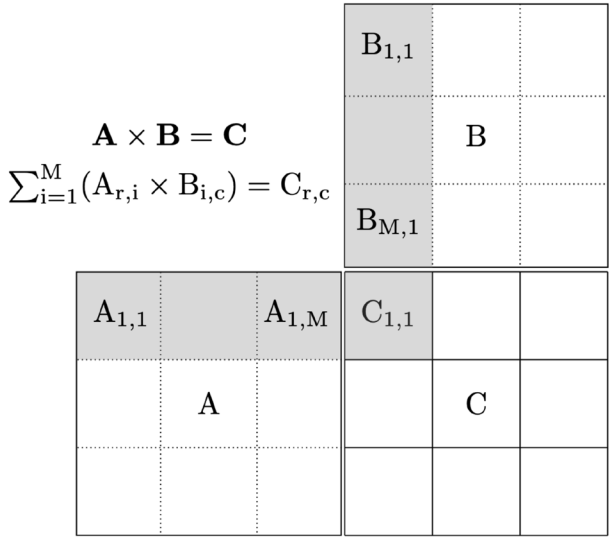
Theoretical example

CPU Matrix Multiplication

CPU Matrix Multiplication

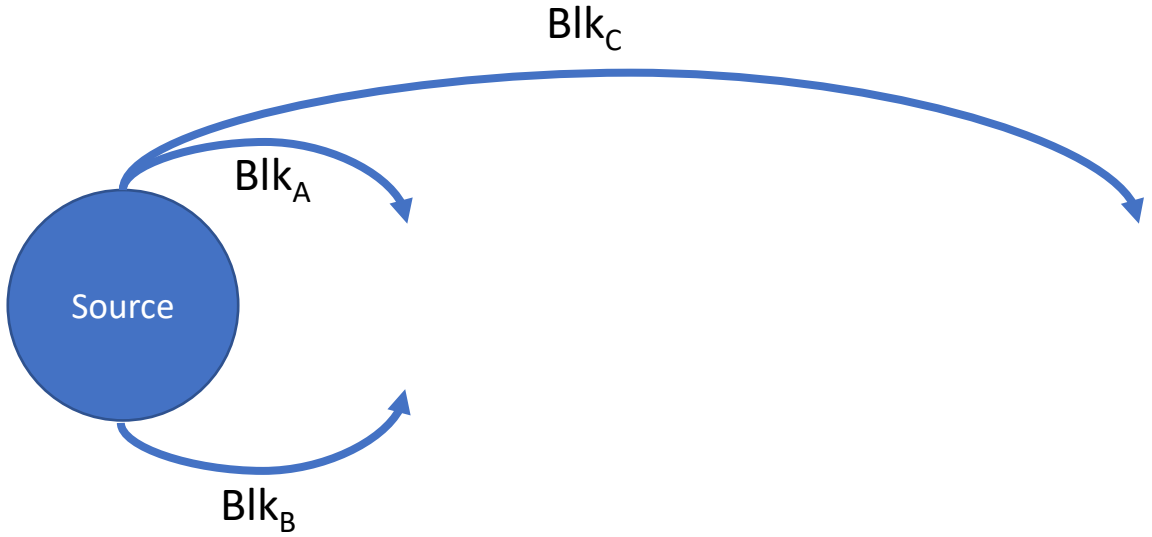
- Matrix Multiplication $C = A \times B + C$
  - Parallel approach:
    - Based on block decomposition → **Input Blocks**
    - Block based partial matrix multiplication (multithreaded) → **Partial Block Result**
    - Aggregation of Partial Block Result (multithreaded) → **Partial Block Result or Final Block Result**
    - Filters Final Block Result → **Final Block Result**

Matrix multiplication decomposition



CPU Matrix Multiplication

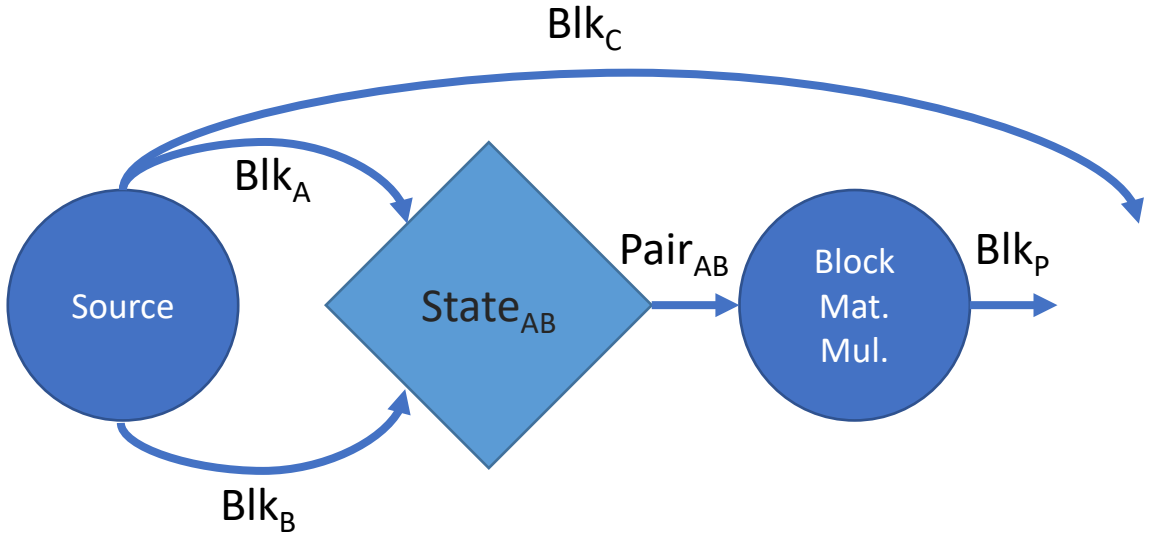
- Matrix Multiplication $C = A \times B + C$
  - Parallel approach:
    - Based on block decomposition → **Input Blocks**
    - Block based partial matrix multiplication (multithreaded) → **Partial Block Result**
    - Aggregation of Partial Block Result (multithreaded) → **Partial Block Result or Final Block Result**
    - Filters Final Block Result → **Final Block Result**



Matrix multiplication decomposition

CPU Matrix Multiplication

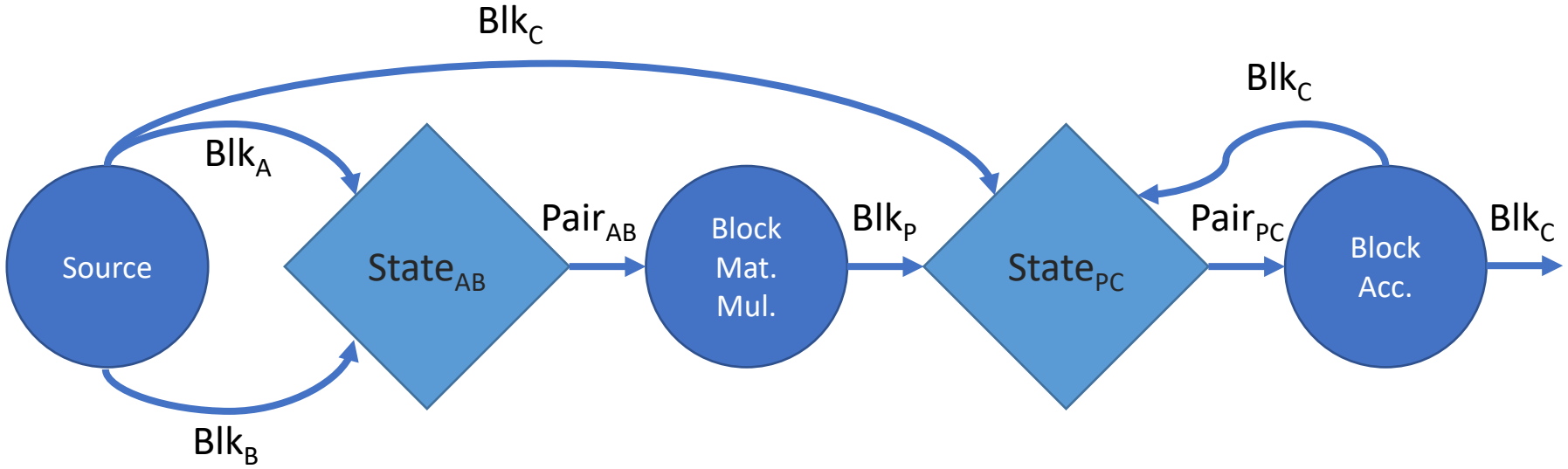
- Matrix Multiplication $C = A \times B + C$
  - Parallel approach:
    - Based on block decomposition → **Input Blocks**
    - Block based partial matrix multiplication (multithreaded) → **Partial Block Result**
    - Aggregation of Partial Block Result (multithreaded) → **Partial Block Result or Final Block Result**
    - Filters Final Block Result → **Final Block Result**



Matrix multiplication decomposition

CPU Matrix Multiplication

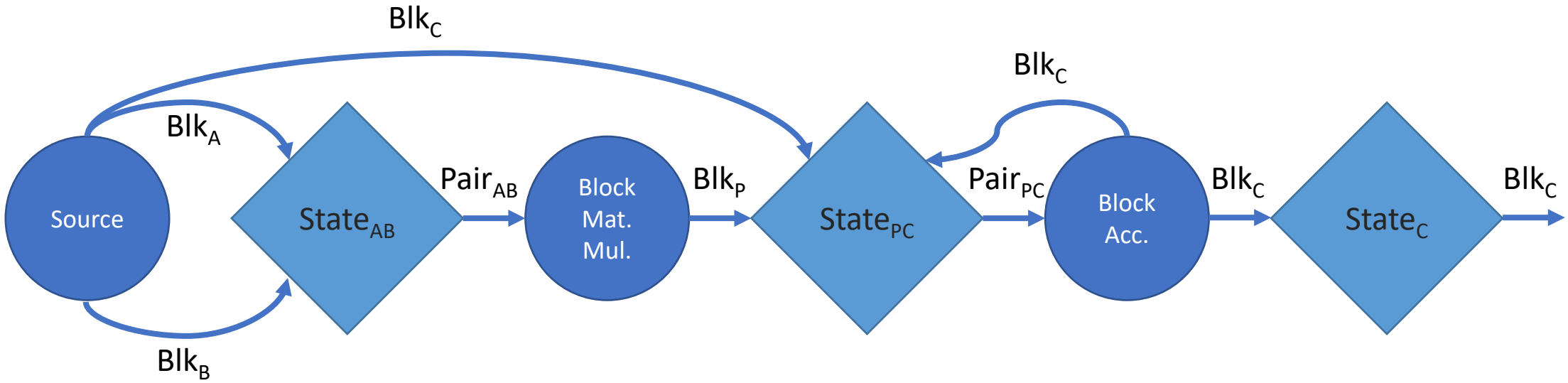
- Matrix Multiplication $C = A \times B + C$
  - Parallel approach:
    - Based on block decomposition → **Input Blocks**
    - Block based partial matrix multiplication (multithreaded) → **Partial Block Result**
    - Aggregation of Partial Block Result (multithreaded) → **Partial Block Result or Final Block Result**
    - Filters Final Block Result → **Final Block Result**



Matrix multiplication decomposition

CPU Matrix Multiplication

- Matrix Multiplication $C = A \times B + C$
  - Parallel approach:
    - Based on block decomposition → **Input Blocks**
    - Block based partial matrix multiplication (multithreaded) → **Partial Block Result**
    - Aggregation of Partial Block Result (multithreaded) → **Partial Block Result or Final Block Result**
    - Filters Final Block Result → **Final Block Result**



Matrix multiplication decomposition

Experiments

Latency

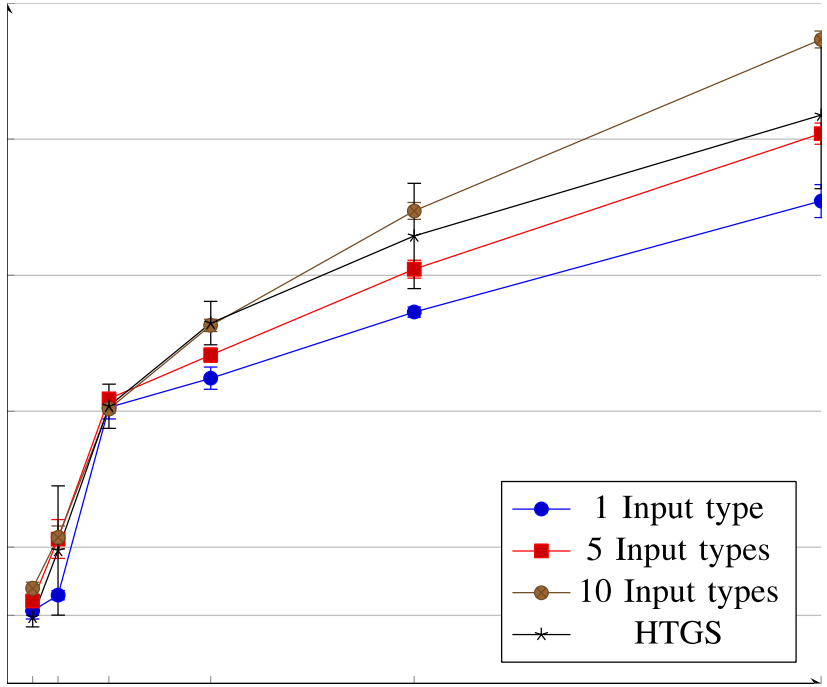
Feedback Profiling

CPU only LU with partial pivoting

Multi-GPU Matrix Multiplication

Latency

- Latency
  - Data transfer between nodes/tasks
- Parameters
  - Number of threads per task
  - Number of input types
    - HTGS can only have 1 input type
- Pipelining hides latency costs
- Hedgehog performance
  - Coarse-grained parallelism
    - Not too big or too small

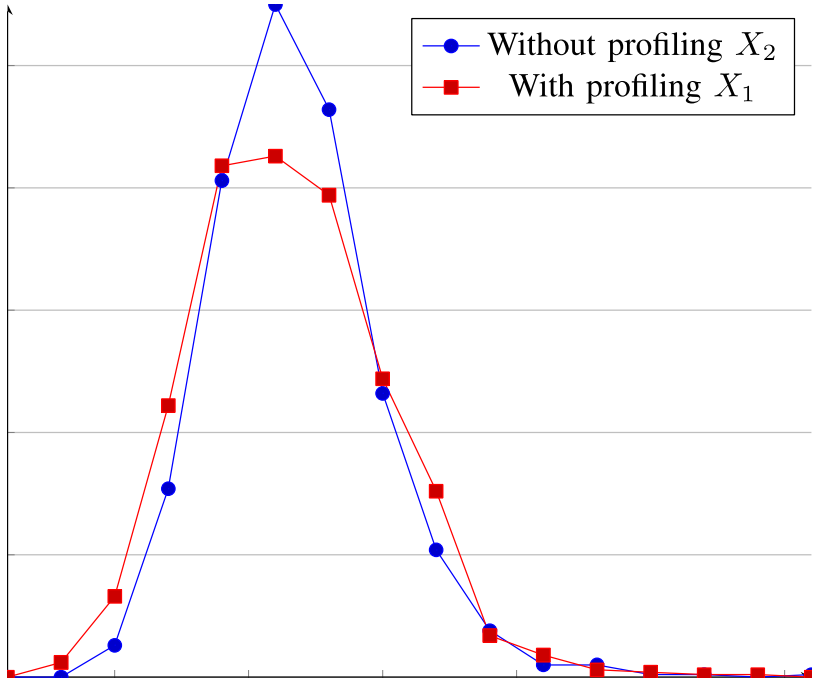


Latency analysis for varying number of threads per task

Profiling cost

- Tutorial 1 (Hadamard product w/ 1 task)
  - 1000 runs
    - with profiling (X_1)
    - without profiling (X_2)
  - $\overline{X_1}$ and $\overline{X_2}$ follow a normal distribution
- H_0: $\mu_1 = \mu_2$
- We pose: $X = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{N} - \frac{s_2^2}{N}}}$

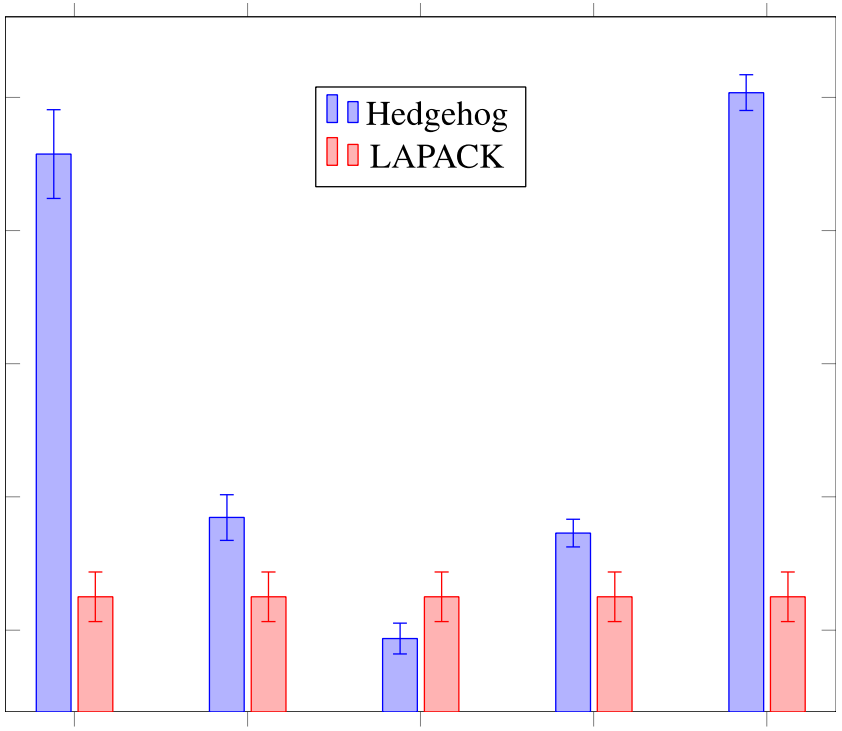
  (standard normal distribution)
- $x = -1{,}32$, p-value $= 0.4066$
- Accept H_0 with a 95% confidence



Execution time distribution for 1000 experiments on $16k \times 16k$ matrices and $2k \times 2k$ blocks with and without profiling

LU with partial pivoting (CPU)

- Methodology allows developers to
  - Reason about complex dependencies
  - Instrument code
- Comparable overall performance to the baseline (LAPACK)
  - When selecting optimal block size
- Get partial fully computed results $40\times$ times faster than entire matrix

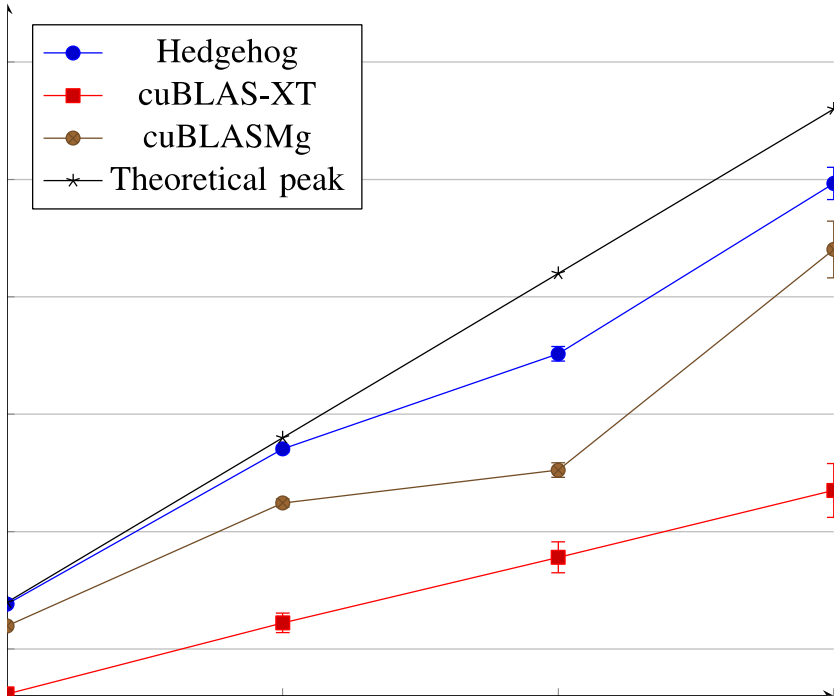


LU with partial pivoting performance for a matrix of $32k \times 32k$ elements

Hardware: Two Xeon E5-2680 v4 @ 2.40 GHz (28 physical cores)
512 GiB DDR4 RAM

Multi-GPU Matrix Multiplication

- Iterative development
  1. Start with CPU only
  2. Augment to GPU
  3. Augment to Multiple GPUs
- Heterogeneous computation
  - Block matrix multiplication on GPUs
  - Block matrix accumulation on CPU
- Task Reuse
  - Tasks that decompose the matrices into blocks
  - State to manage the computation
  - GPU computation and I/O graph



Matrix multiplication on a matrix of $64k \times 64k$ elements decomposed in $8k \times 8k$ block elements

Hardware: Two Xeon Silver 4216 @ 2.10 GHz (32 physical cores)
768 GiB DDR4 RAM, 4x Tesla V100 PCIe with 32 GB HBM2

Conclusion

How to develop an **understandable parallel** program/algorithm on **heterogeneous** nodes?

- Explicit data-flow graph representation
  - Design = Execution
- Costless graphical feedback
  - Understand the computation
  - Detect bottlenecks
    - Gain insight
- API based on separation of concerns
  - Distinct components:
    - Compute vs. state manager
  - Graph composability / reuse
  - Target heterogeneous nodes / Memory locality
- Results
  - Partial result streaming
  - Portable design for performance

Future work

- Extend Hedgehog to clusters
  - External libraries/tools for cluster communication
  - One Hedgehog graph per cluster node

- Static Analysis of the graph
  - Run at compile time algorithm to test further the graph
  - Detect possible race conditions, cycles, or deadlocks

Thank you

Contact us for more information

- Alexandre Bardakoff (alexandre.bardakoff@nist.gov)
- Timothy Blattner (timothy.blattner@nist.gov)

Get the code and tutorials

- Hedgehog:
  - https://github.com/usnistgov/hedgehog
- Tutorials:
  - https://pages.nist.gov/hedgehog-Tutorials
  - https://github.com/usnistgov/hedgehog-Tutorials