SAND2022-6922C

Sandia National Laboratories

Machine Learning for CUDA+MPI Design Rules

Carl Pearson, Karen Devine, Aurya Javeed

Sandia National Labs

PDSEC 2022

This work is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute.
This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award ERCAP0019623.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Automatic Discovery of Implementation Rules for Fast GPU + MPI Operations

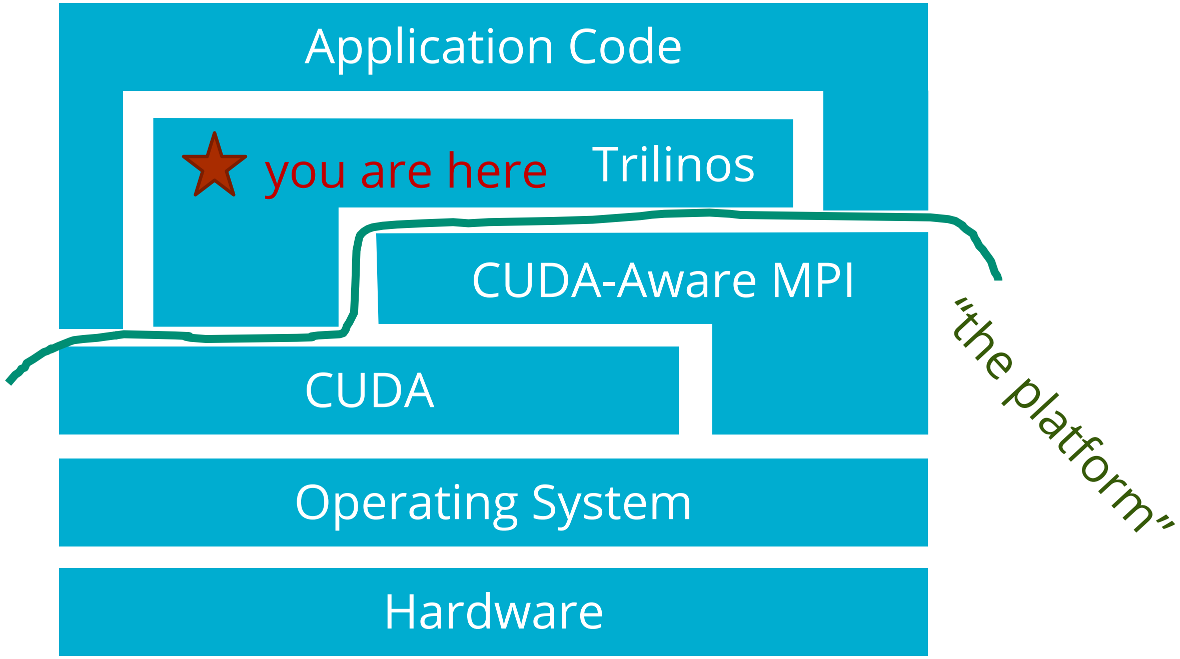
- Fast libraries for heterogeneous architectures
  - Mapping computation onto processors
  - Choosing communication strategy
  - Unpredictable performance interaction
- Prototype automatic tooling for discovering important design decisions
  - Reduced developer effort for performance on new systems
  - Maintain human provenance of library design
  - e.g. Modernize Tpetra MPI+GPU distributed linear algebra operations

Key Challenge	How it's Done
Large Design Space	• Express operation as a directed acyclic graph (DAG) of operations • Monte-Carlo Tree Search (MCTS) to identify and explore regions of interest
Extract performance insight	• Empirical benchmarking • Feature vector for each implementation • Decision tree training for design rules

Initial results pass “sniff test,” working on broader experiments and quantitative evaluation

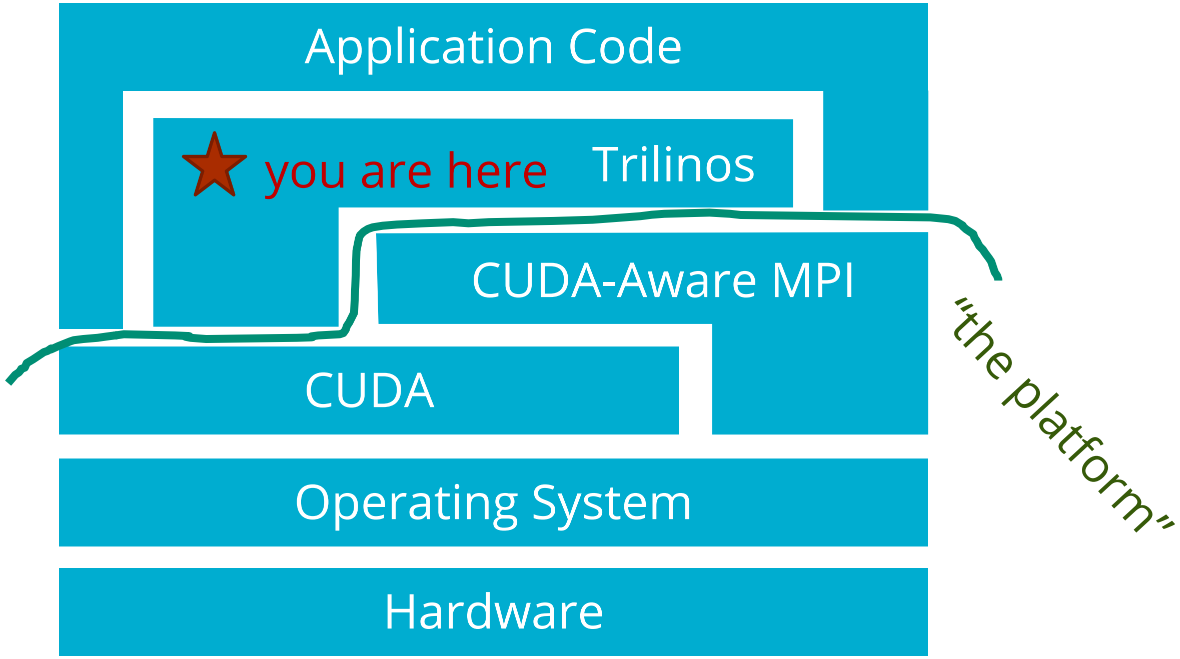
Libraries are built on existing lower-level primitives

- Our libraries (and applications) are combinations of existing library and vendor operations
  - and code to coordinate them
  - and code to implement custom behavior

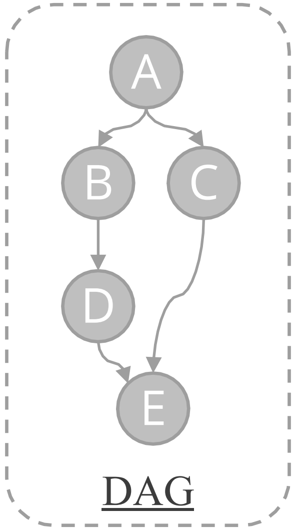


Libraries are built on existing lower-level primitives

- Our libraries (and applications) are combinations of existing library and vendor operations
  - and code to coordinate them
  - and code to implement custom behavior
- Performance changes at many layers for new platforms
  - new hardware,
  - new CUDA version,
  - new OS version,
  - etc.



Prototype Implementation in C++ and Python

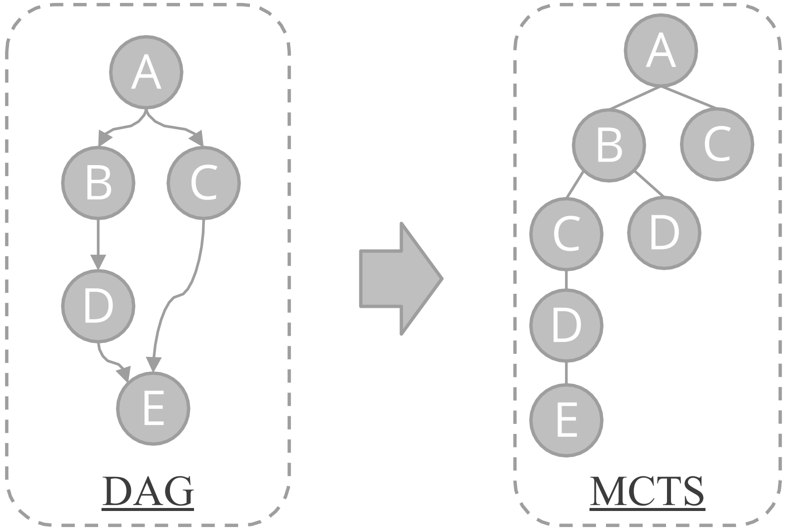


DAG of operations describes design space

C++ / CUDA / MPI

Python / scikit-learn

Prototype Implementation in C++ and Python



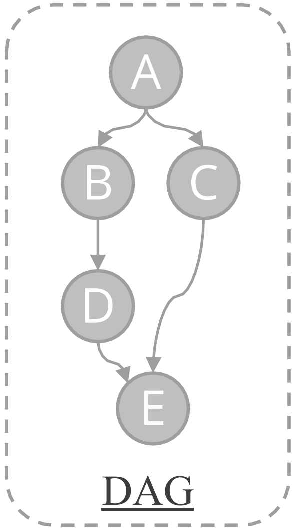
DAG of operations describes design space

MCTS searches order of operations and resource assignment

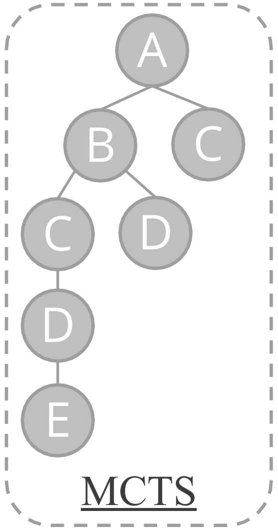
C++ / CUDA / MPI

Python / scikit-learn

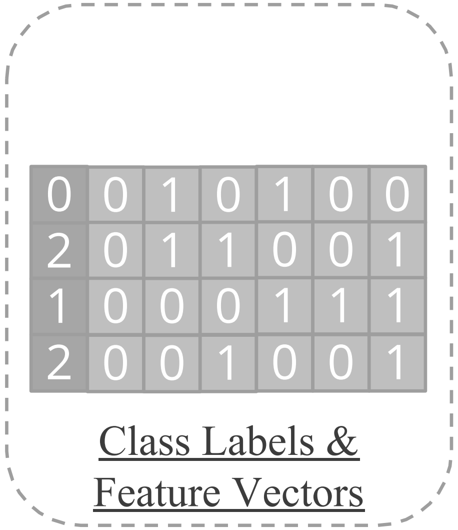
Prototype Implementation in C++ and Python



DAG of operations describes design space



MCTS searches order of operations and resource assignment

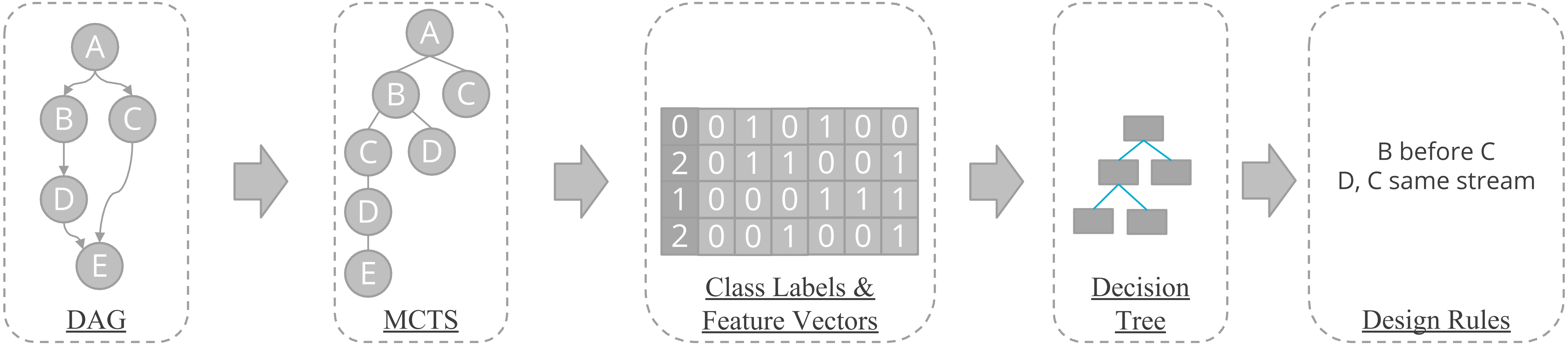


Sequence-to-vector transformation for labels

C++ / CUDA / MPI

Python / scikit-learn

Prototype Implementation in C++ and Python



DAG of operations describes design space

MCTS searches order of operations and resource assignment

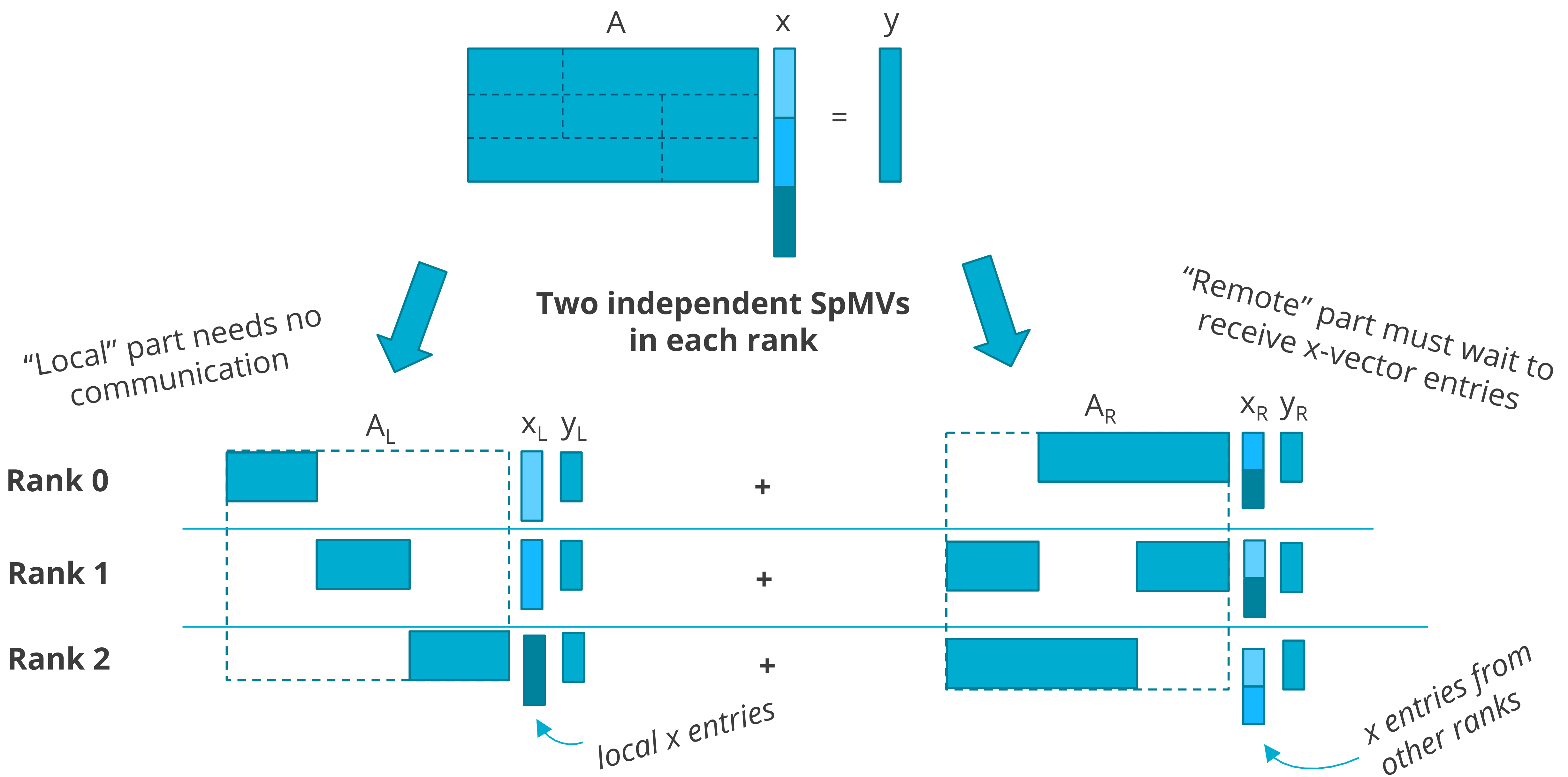
Sequence-to-vector transformation for labels

Decision tree training to identify discriminating design space features

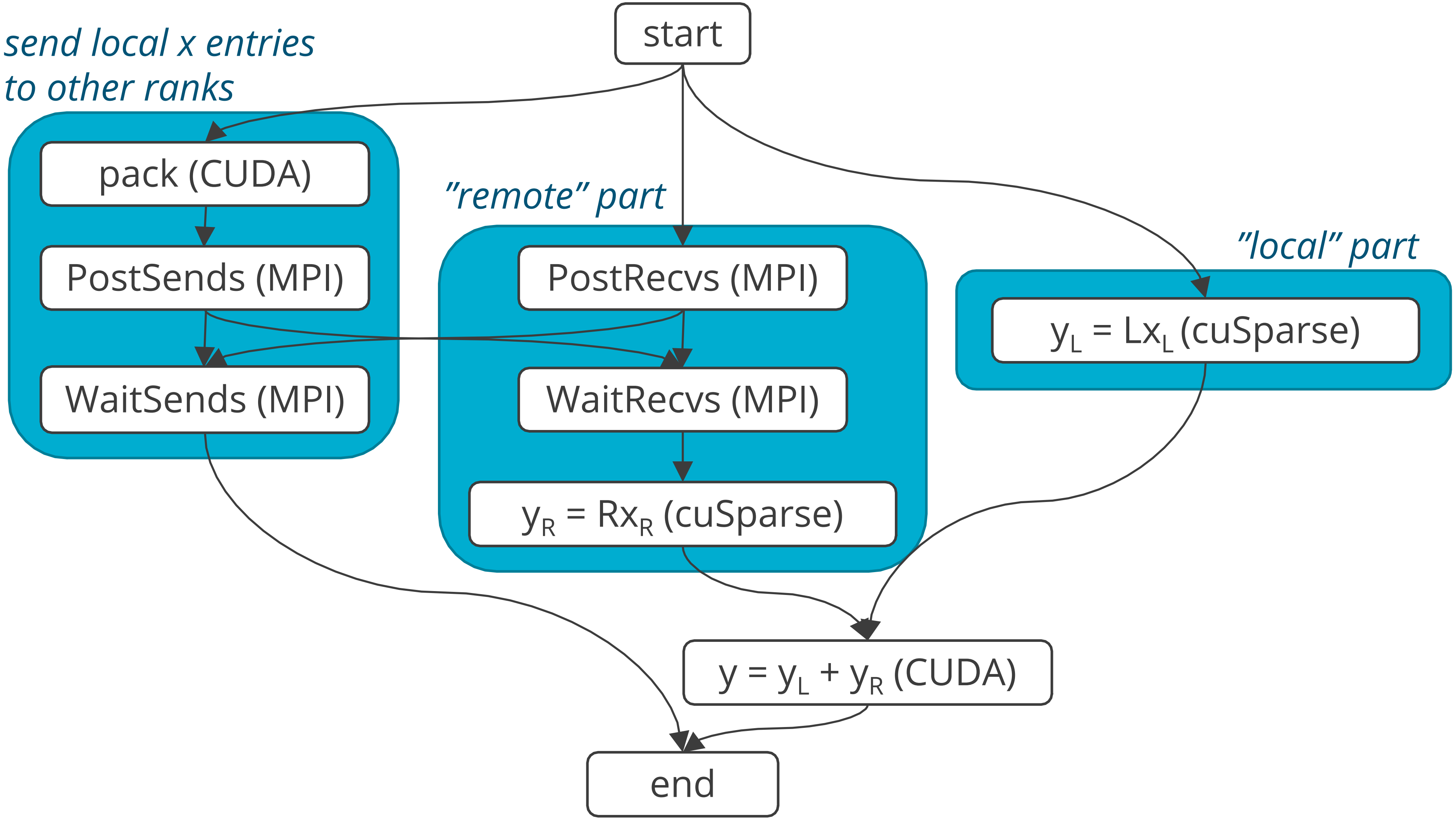
C++ / CUDA / MPI

Python / scikit-learn

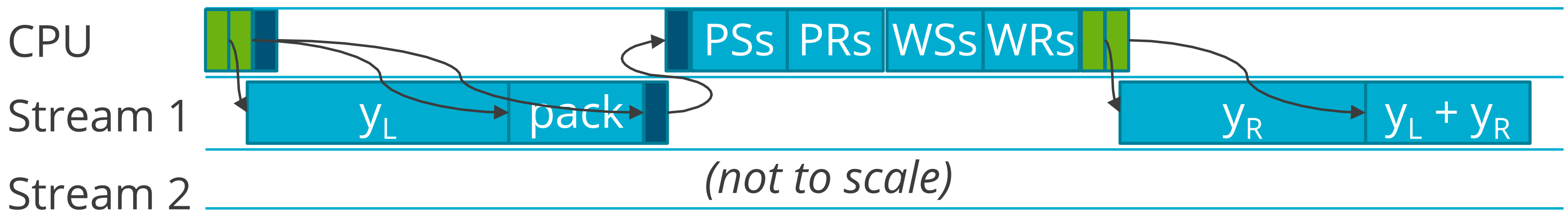
Example: Distributed SpMV



DAG represents primitive operations and their dependences

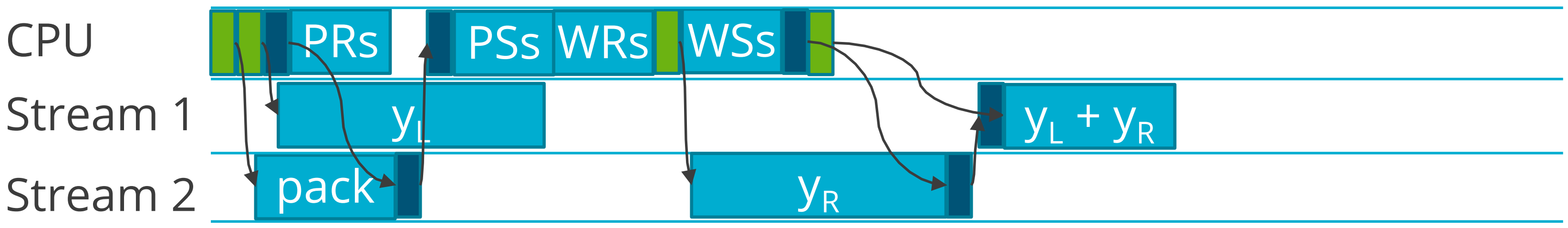


Design Space: Order of Operations, Resource Assignment, and Synchronization



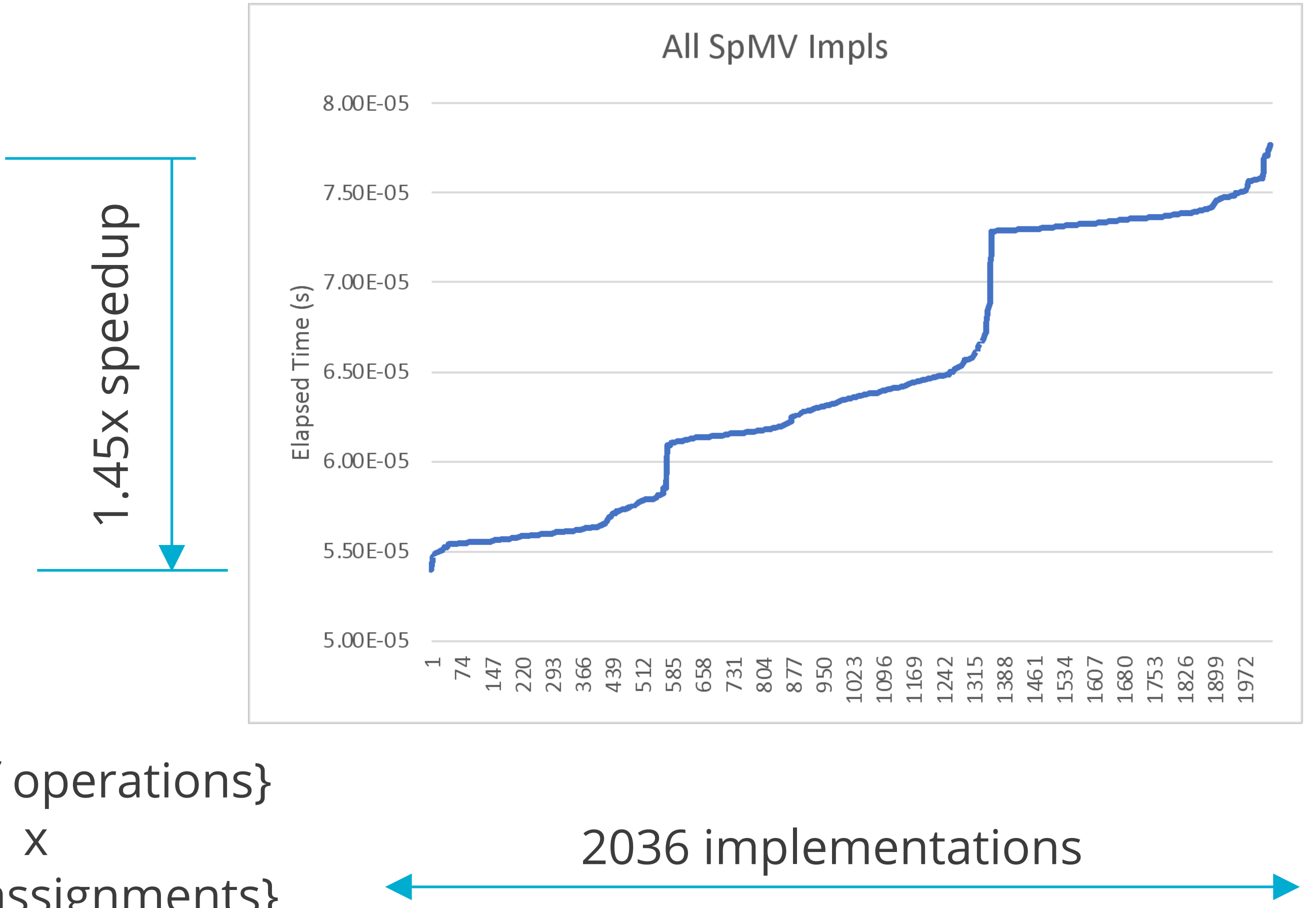
- Different resource assignments require different synchronization
- May improve GPU utilization or communication/computation overlap, but increases required operations

kernel launch
sync ops
application operations



Need to Discover Important Design Decisions

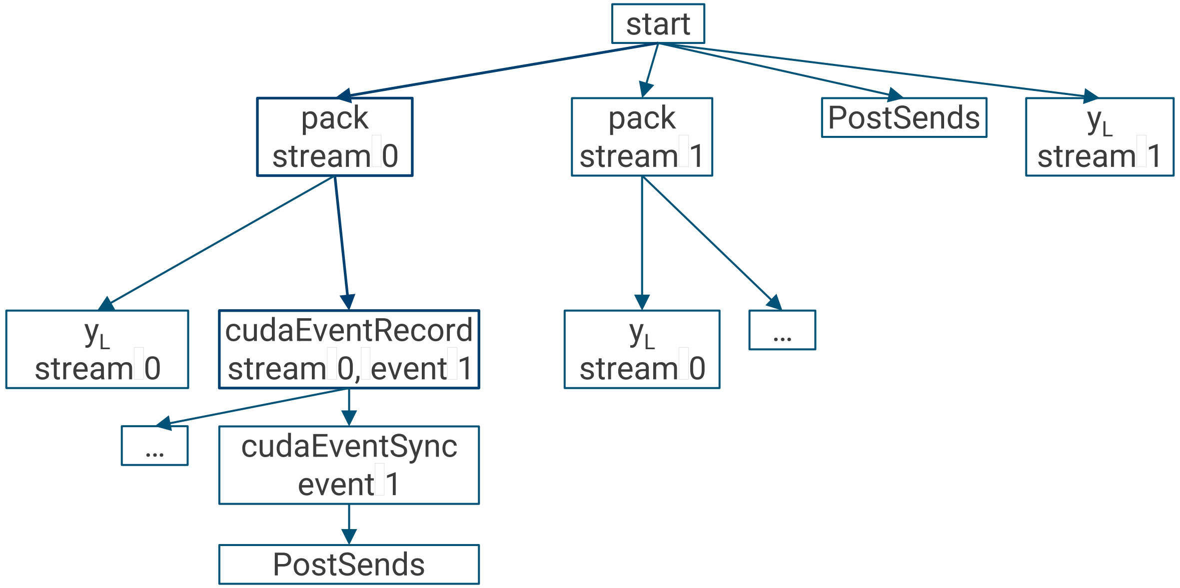
- Some choices matter a lot
- Many choices do not matter at all
- input- and system-dependent
- Large design space: lots of expert time to evaluate and implement for each target platform
- Monte-Carlo Tree Search to focus on valuable decisions



{order of operations}
x
{stream assignments}
x
{synchronizations}

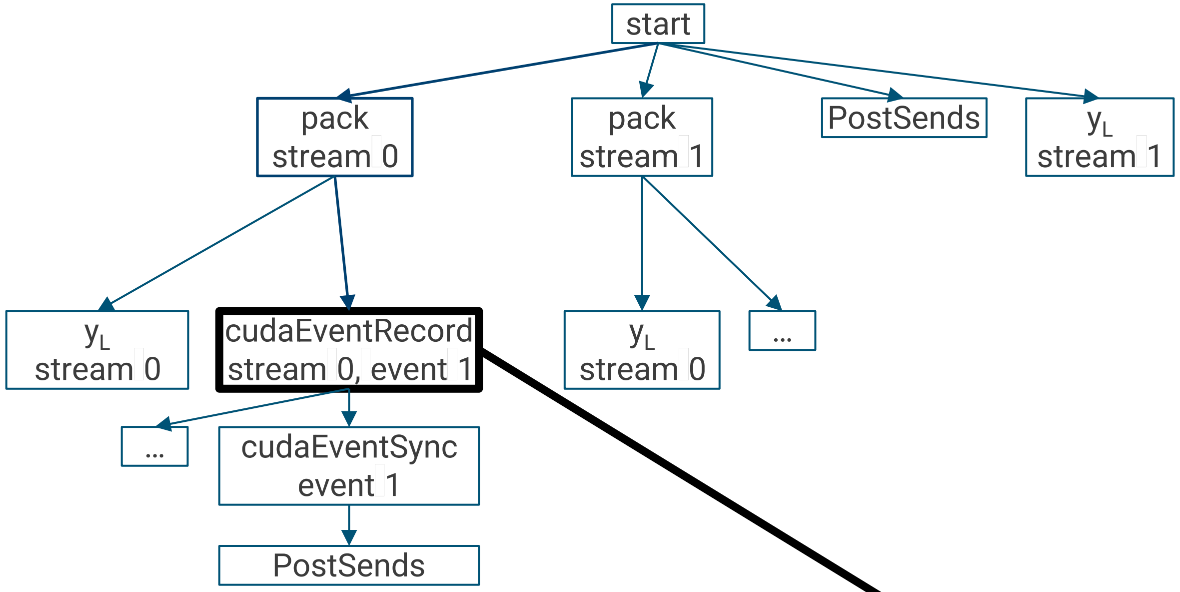
MCTS Represents Search State in a Tree

State space search is stored in a tree



MCTS Represents Search State in a Tree

State space search is stored in a tree



Each node is an operation and resource assignment

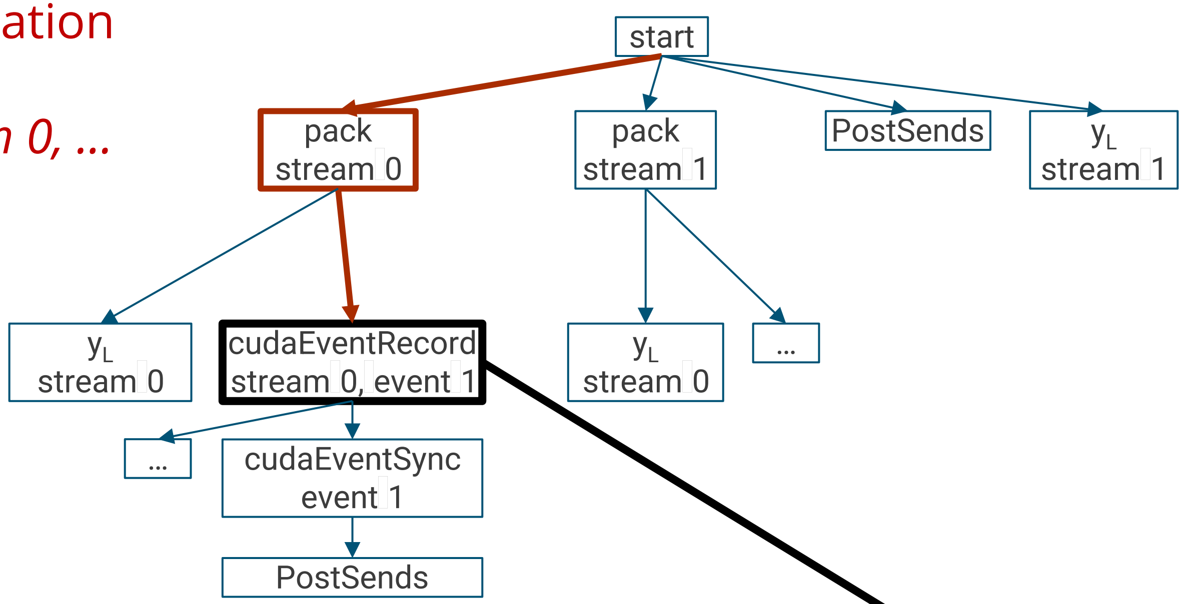
From DAG, or synchronization operation

MCTS Represents Search State in a Tree

State space search is stored in a tree

Path is the beginning of an implementation

pack in stream 0, record event 1 in stream 0, …



Each node is an operation and resource assignment

From DAG, or synchronization operation

MCTS Represents Search State in a Tree

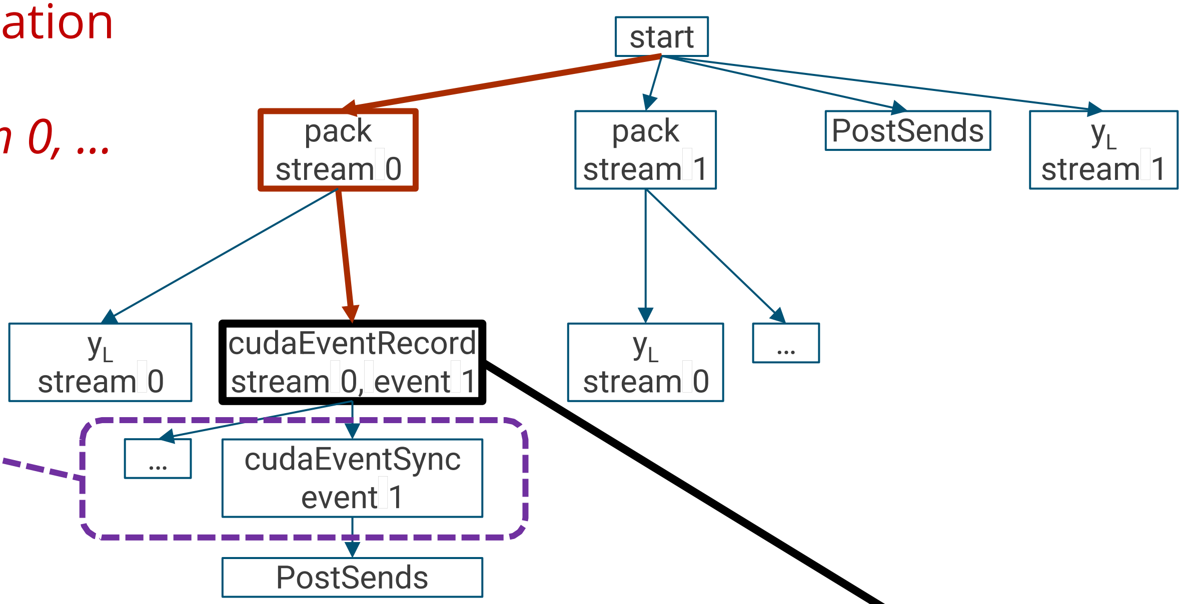
State space search is stored in a tree

Path is the beginning of an implementation

pack in stream 0, record event 1 in stream 0, …

Children are all possible subsequent operation / resource combinations

All DAG predecessors complete and synchronized



Each node is an operation and resource assignment

From DAG, or synchronization operation

MCTS Represents Search State in a Tree

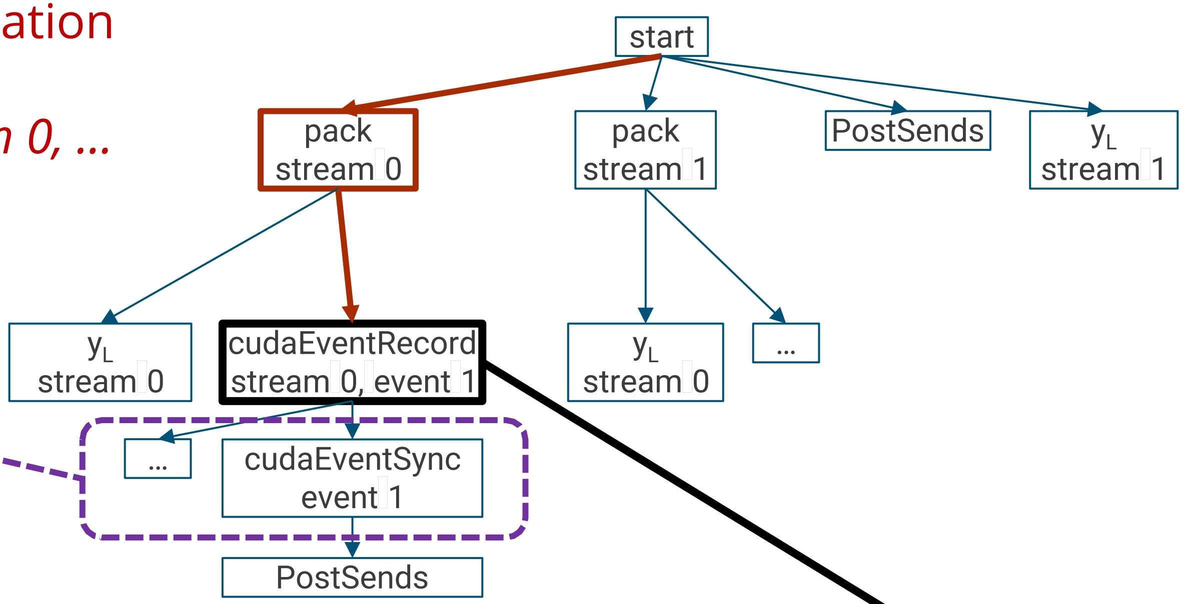
State space search is stored in a tree

Path is the beginning of an implementation

pack in stream 0, record event 1 in stream 0, …

Children are all possible subsequent operation / resource combinations

All DAG predecessors complete and synchronized



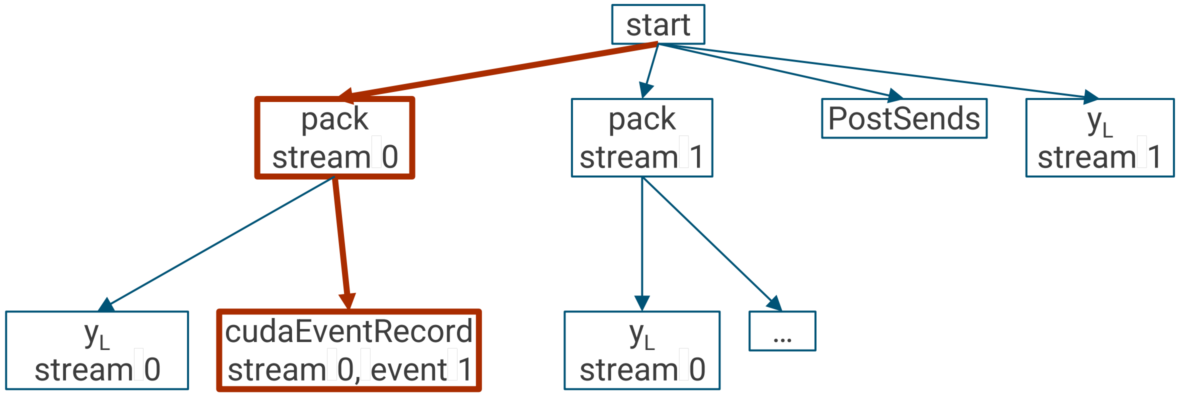
Each node stores empirical performance of any complete implementation it is part of

Each node is an operation and resource assignment

From DAG, or synchronization operation

MCTS Iteratively Grows Tree to Focus on Valuable Regions

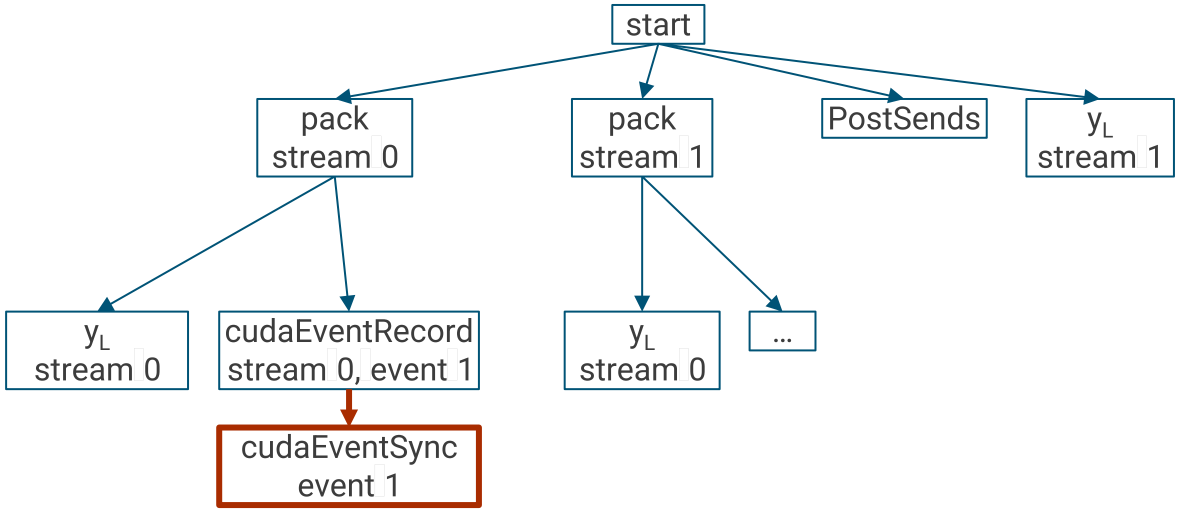
Selection: Choose a path through the tree, balancing valuable vs unexplored subtrees



MCTS Iteratively Grows Tree

Selection: Choose a path through the tree, balancing valuable vs unexplored subtrees

Expansion: Create a new child

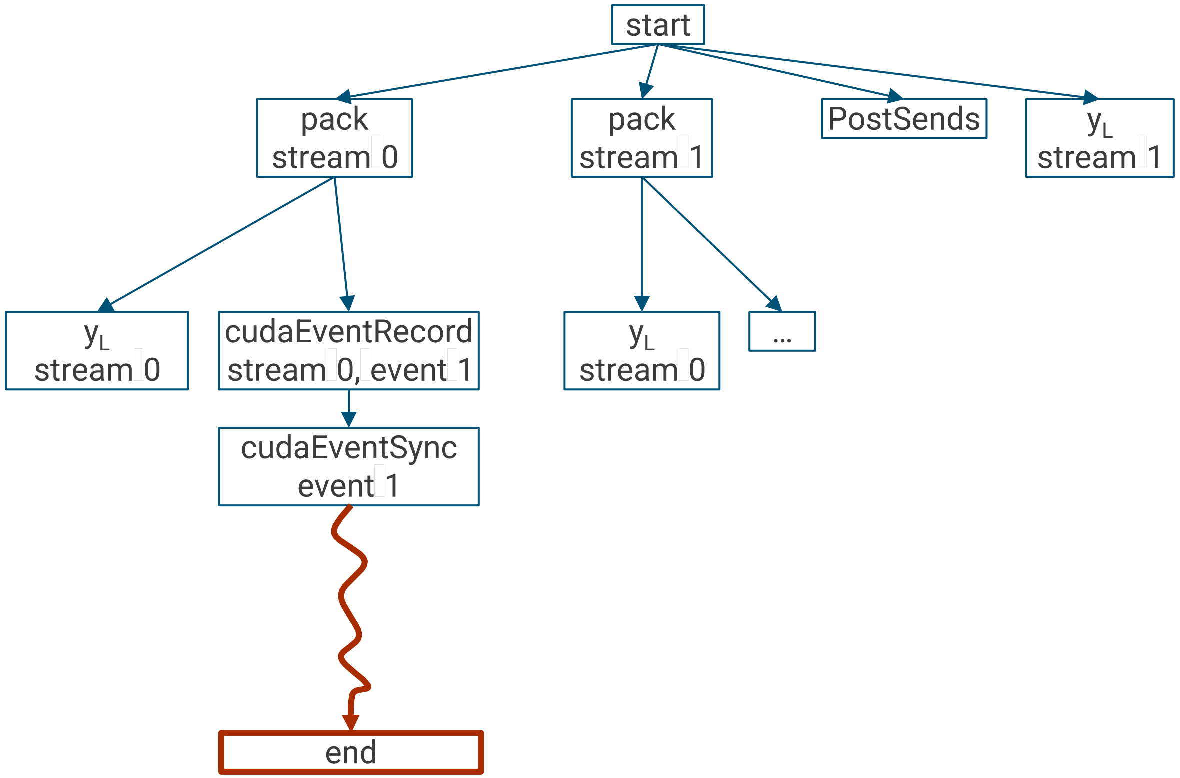


MCTS Iteratively Grows Tree

Selection: Choose a path through the tree, balancing valuable vs unexplored subtrees

Expansion: Create a new child

Rollout: Random ordering / assignment to complete the implementation



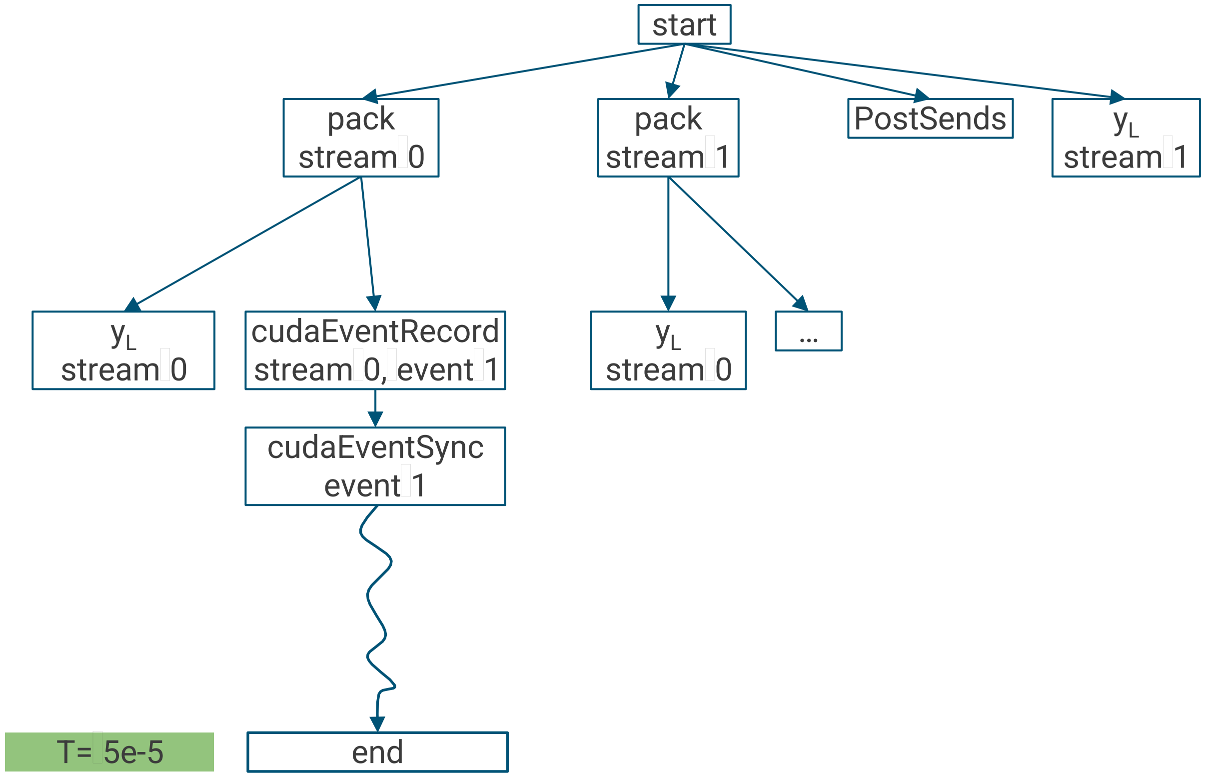
MCTS Iteratively Grows Tree

Selection: Choose a path through the tree, balancing valuable vs unexplored subtrees

Expansion: Create a new child

Rollout: Random ordering / assignment to complete the implementation

Evaluation: Empirical benchmark



MCTS Iteratively Grows Tree

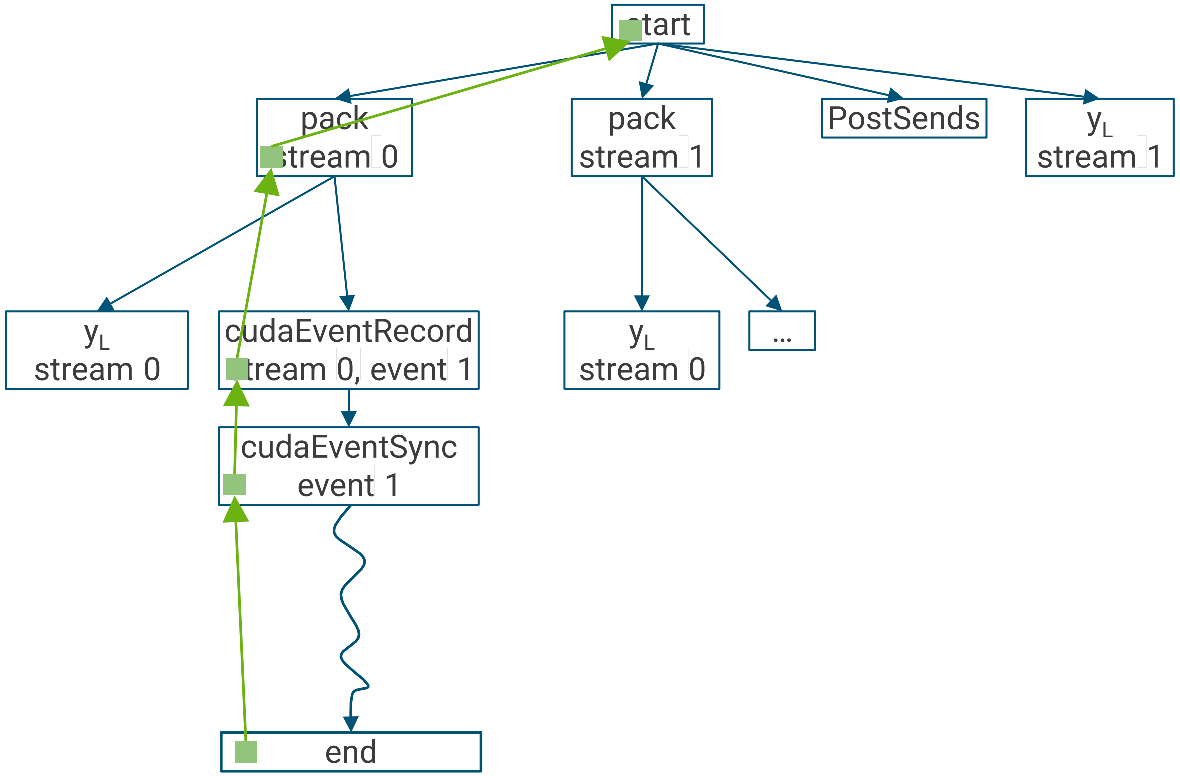
Selection: Choose a path through the tree, balancing valuable vs unexplored subtrees

Expansion: Create a new child

Rollout: Random ordering / assignment to complete the implementation

Evaluation: Empirical benchmark

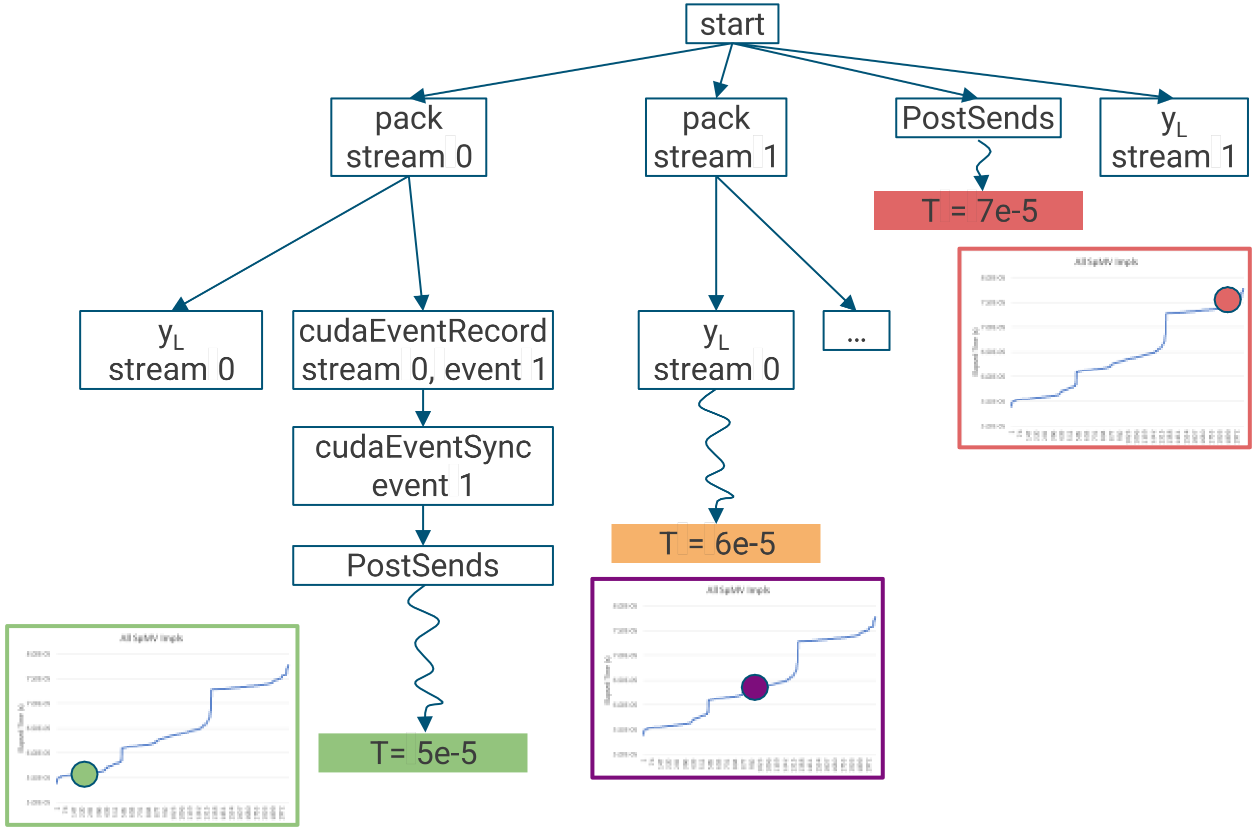
Backpropagation: Update each node with new empirical result



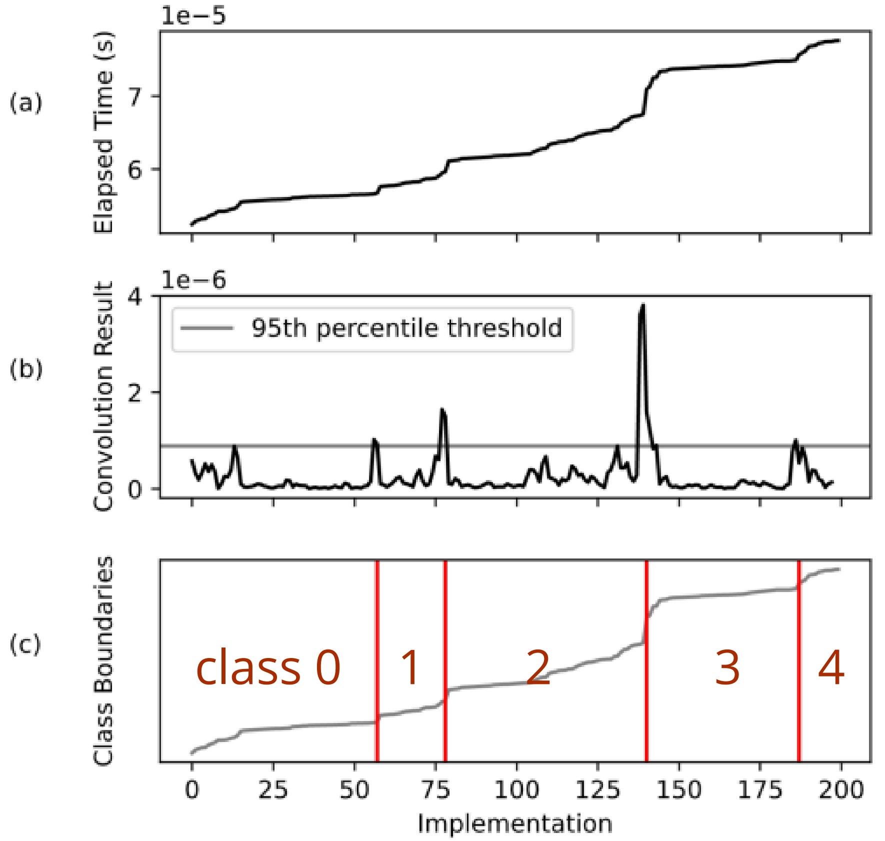
Tree is Deeper and Larger in Valuable Regions

As iterations proceed, tree preferentially explores high-reward regions of the design space

Store all complete implementations and performance results in a table as we go



Transform Empirical Results into Performance Classes and Feature Vectors



subset explored by MCTS

ordering rules | *resource assignment rules*

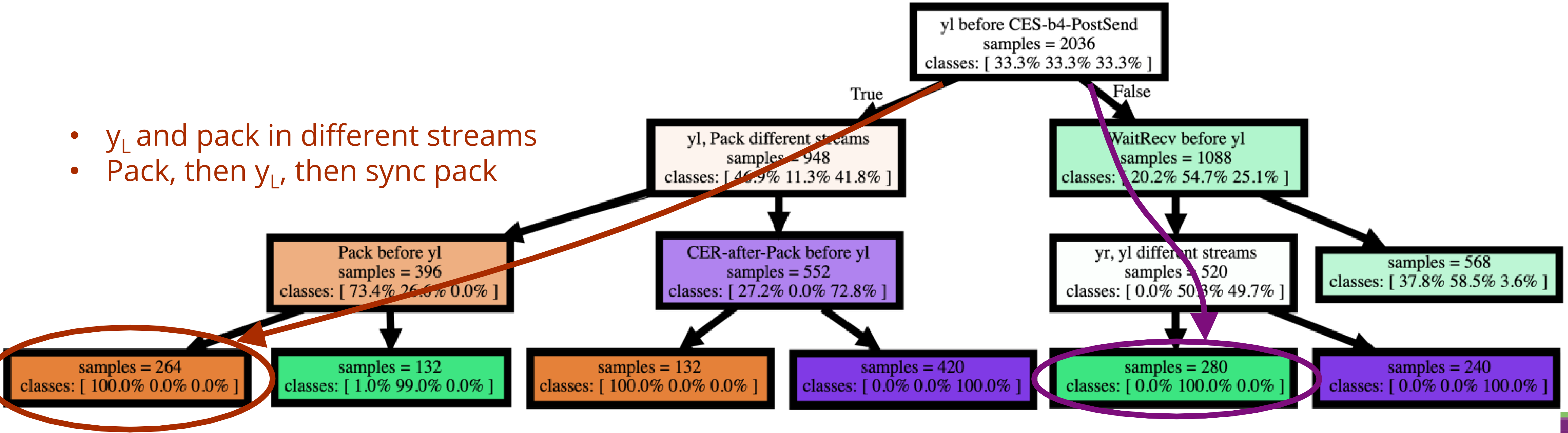
Impl.	Class Label	A then B	…	A same stream B	…
98	2	0	1	1	0
0	0	1	0	1	0
56	1	0	0	1	1
73	1	0	0	1	1
…					…

automatic class labeling to identify performance classes (convolution & peak detection)

feature vectors encode which rules an implementation follows (sequence-to-vector transformation)

Decision Tree Training to Determine which Rules Discriminate between Classes

- y_L and pack in different streams
- Pack, then y_L, then sync pack

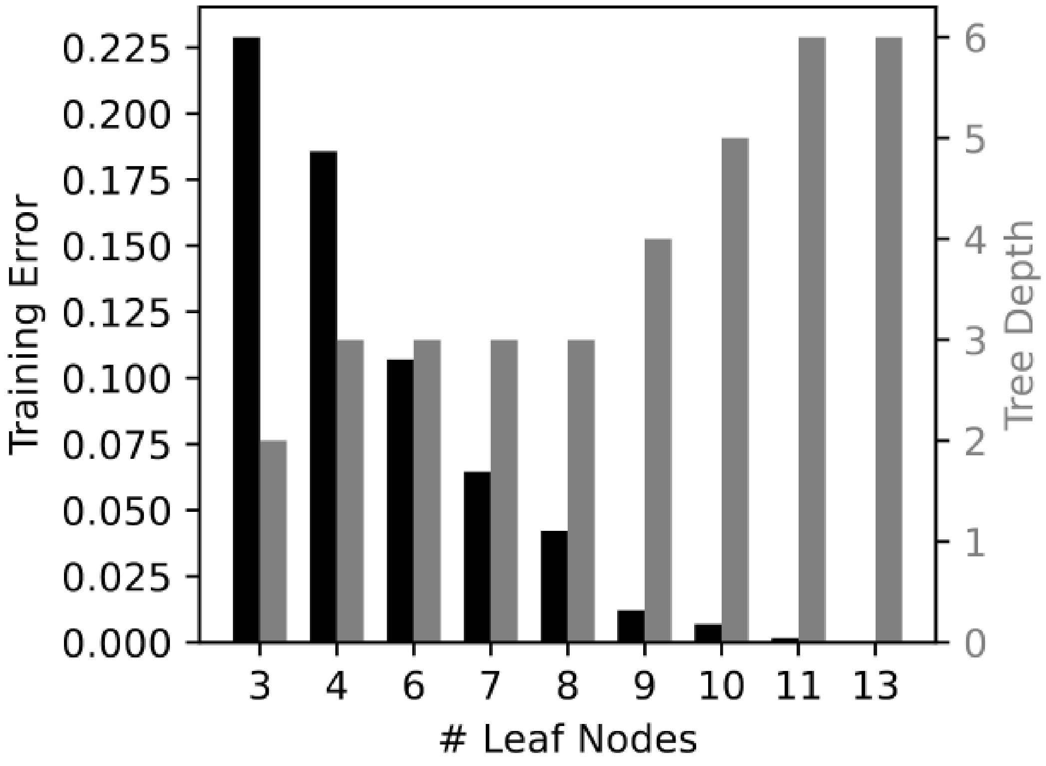


- sync pack before y_L
- WaitRecv before y_L
- y_L, y_R in same stream

Each path through the tree is a set of design rules that define a performance class

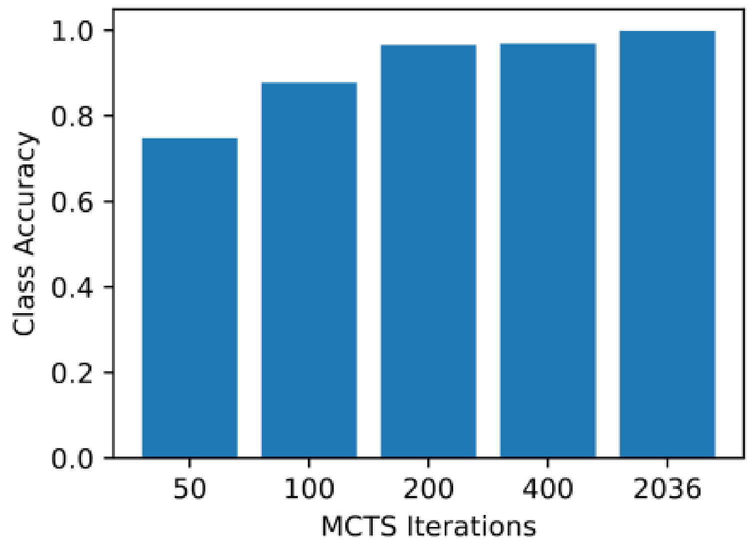
Train an Accurate Decision Tree

- Training process is for isolating discriminating features
  - **not** for classifying unseen inputs
- Incrementally increase tree size until 100% accuracy achieved
- Accuracy-complexity tradeoff in generated rules



Does MCTS Find Relevant Design Space Regions?

- Each MCTS iteration is a costly empirical benchmark
- Rule quality with reduced iterations?
  - For a given # of iterations, how accurate are the rules?
  - For a given # of iterations, qualitative look at the rules?



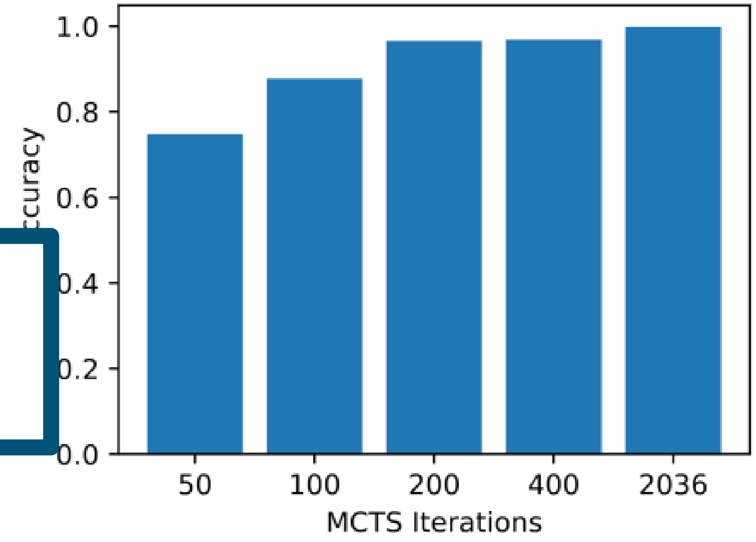
MCTS Iterations	2036	50	100	200	400
Discovered Ruleset for Fastest Performance Class	$y_L \rightarrow$ CES-b4-PostSend y_L ╳ Pack Pack $\rightarrow y_L$	$y_L \rightarrow$ CES-b4-PostSend y_L ╳ Pack Pack $\rightarrow y_L$	$y_L \rightarrow$ CES-b4-PostSend y_L ╳ Pack Pack $\rightarrow y_L$ $y_L \rightarrow$ WaitSend	$y_L \rightarrow$ CES-b4-PostSend y_L ╳ Pack Pack before y_L $y_L \rightarrow$ WaitSend	$y_L \rightarrow$ WaitRecv PostSend $\rightarrow y_L$ Pack $\rightarrow y_L$ CER-after-Pack $\rightarrow y_L$ $y_L \rightarrow$ WaitSend PostRecv $\rightarrow$ CES-b4-PostSend

A ╳ B: A different stream than B
A $\rightarrow$ B: A, then B

Most populous ruleset shown

Does MCTS Find Relevant Design Space Regions?

- Each MCTS iteration is a costly empirical benchmark
- Rule quality with reduced iterations?
  - For a given ≠
  - For a given ≠



Few iterations → approx. random sample
Sample distribution = exhaustive search

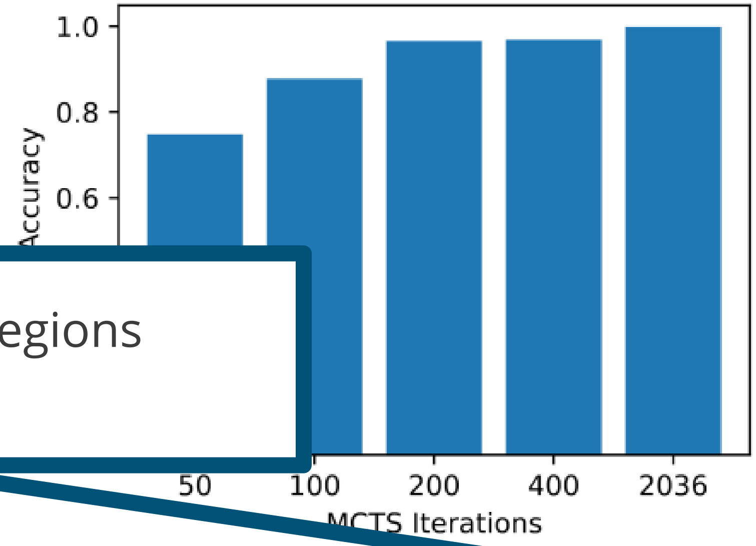
MCTS Iterations	2036	50	100	200	400
Discovered Ruleset for Fastest Performance Class	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L y_L → WaitSend	y_L → CES-b4-PostSend y_L ╳ Pack Pack before y_L y_L → WaitSend	y_L → WaitRecv PostSend → y_L Pack → y_L CER-after-Pack → y_L y_L → WaitSend PostRecv → CES-b4-PostSend

A ╳ B: A different stream than B
A → B: A, then B

Most populous ruleset shown

Does MCTS Find Relevant Design Space Regions?

- Each MCTS iteration is a costly empirical benchmark
- Rule quality with reduced iterations?
  - Fo
  - Fo



More iterations → samples drawn from valuable regions
More samples fall into different rules

MCTS Iterations	2036	50	100	200	400
Discovered Ruleset for Fastest Performance Class	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L	y_L → CES-b4-PostSend y_L ╳ Pack Pack → y_L y_L → WaitSend	y_L → CES-b4-PostSend y_L ╳ Pack Pack before y_L y_L → WaitSend	y_L → WaitRecv PostSend → y_L Pack → y_L CER-after-Pack → y_L y_L → WaitSend PostRecv → CES-b4-PostSend

A ╳ B: A different stream than B
A → B: A, then B

Most populous ruleset shown

Vision for this work

- Current
  - C++ MCTS implementation for MPI/CUDA codes with multiple streams
  - Prototype feature-vector and decision tree training using SciKit in Python
  - Available in March at github.com/sandialabs/tenzing-core
- Upcoming
  - Applying initial results to Tpetra distributed linear algebra package in Trilinos
- Future Explorations
  - Identify unexpected performance effects on target platforms ("performance bugs")
  - What to do as communication / computation are more tightly integrated
- Summary
  - Represent CUDA+MPI operation as DAG
  - Automatically generate human-interpretable rules for library design
  - Maintain human provenance of implementation (no "black boxes")