Efficient Computation of Map-scale Continuous Mutual Information on Chip in Real Time

Keshav Gupta, Peter Z. X. Li, Sertac Karaman, Vivienne Sze


Published in IEEE IROS 2021|Sponsored by NSF RTML 1937501 and NSF CPS 1837212
Overview

- Autonomous robotic exploration has applications in deep sea exploration and mapping, space exploration, search and rescue, and more!
• Algorithms used to decide an efficient path are compute intensive!
  – Infeasible for low- and medium-power robots.
Overview

• **Data dependencies** in the algorithm challenge efficient hardware design.
Overview

• **Data dependencies** in the algorithm challenge efficient hardware design.

We propose a **real-time** accelerator that is able to tackle these challenges!
Contents

• Problem Description
  – What is robotic exploration?
  – Alternate exploration algorithms
Contents

• Problem Description
  – What is robotic exploration?
  – Alternate exploration algorithms

• Algorithm chosen: Fast Continuous Mutual Information
Contents

• Problem Description
  – What is robotic exploration?
  – Alternate exploration algorithms

• Algorithm chosen: Fast Continuous Mutual Information

• Hardware Description
  – Challenges faced, and solutions
Contents

• Problem Description
  – What is robotic exploration?
  – Alternate exploration algorithms

• Algorithm chosen: Fast Continuous Mutual Information

• Hardware Description
  – Challenges faced, and solutions

• Evaluation of design, comparison with other platforms
  – Latency
  – Energy Consumption
  – Impact on exploration efficiency
What is robotic exploration?

- Goal: create a map of an unknown environment
What is robotic exploration?

• Goal: create a map of an unknown environment

• Input: Radial range measurements from on-board sensor
  – E.g. LiDAR, SONAR
What is robotic exploration?

- **Goal:** create a map of an unknown environment
- **Input:** Radial range measurements from on-board sensor
  - E.g. LiDAR, SONAR
- **Output:** Complete map of the environment
  - In this application, represented as an Occupancy Grid Map
The process of exploration

• Until a complete occupancy grid map is obtained, repeat the following:

  Make range measurement
The process of exploration

- Until a complete occupancy grid map is obtained, repeat the following:

  1. Make range measurement
  2. Select next location
The process of exploration

- Until a complete occupancy grid map is obtained, repeat the following:

  1. Make range measurement
  2. Select next location
  3. Move to next location
Example exploration

Occupancy Grid with planned paths
- Occupied Space
- Unknown Space
- Free Space

Calculated Mutual Information
Brighter colors = higher MI
Next location selection algorithms

• Consider this example scenario:

How do we select the next location to move to?
Next location selection algorithms

• Choice affects the efficiency of exploration:
  – Trajectory length
  – Time taken
  – Energy consumed
Choice affects the efficiency of exploration:
  – Trajectory length
  – Time taken
  – Energy consumed

Make-or-break for low-power robots!
Next location selection algorithms

• Consider this example scenario:

How do we select the next location to move to?
What is Mutual Information (MI)?

- Amount of potential information gain at a scan location:

Move to location with the **highest MI!**
What is Mutual Information (MI)?

- Amount of potential information gain at a scan location:

Move to location with the **highest MI**!
What is Mutual Information (MI)?

• We see MI being calculated in the bottom right map here:
Formulating MI

- Fast Shannon Mutual Information (FSMI) \(^1\) is one of the algorithms used to compute MI.

Formulating MI

• Fast Shannon Mutual Information (FSMI) \(^{(1)}\) is one of the algorithms used to compute MI.

• It is highly compute intensive:
  – Even a hardware accelerator \(^{(2)}\) is not real time (sensor rate: 30-60 Hz).

---


Formulating MI

• Fast Shannon Mutual Information (FSMI) [1] is one of the algorithms used to compute MI.

• It is highly compute intensive:
  – Even a hardware accelerator [2] is not real time (sensor rate: 30-60 Hz).

• Fast Continuous Mutual Information (FCMI) [3] reduces compute complexity.
  – Introduces data dependencies which complicate parallel compute.

Formulating MI

- Compute the MI map, then plan the path:

  ![Diagram showing MI map and path planning process](image-url)
Formulating MI

• Compute the MI map, then plan the path:

  ![MI map](image)

  ➔

  ![Path planner](image)

• How do you compute it?

• Why is it so compute intensive?
• Let's say we want to compute the MI at:
Computing FSMI

• Let us look at an individual range measurement ray:
Computing FSMI

- MI contribution of a ray is a function of cell occupancy values:
Computing FSMI

• MI contribution of a ray is a function of cell occupancy values:

\[
MI_{\text{ray}} = \sum_{i=1}^{\text{ray length}} \Pr(\text{ray hits cell } i) \times \text{MI from cell } i
\]
Computing FSMI

- MI contribution of a ray is a function of cell occupancy values:

\[ MI_{\text{ray}} = \sum_{i=1}^{\text{ray length}} \Pr(\text{ray hits cell } i) \times \text{MI from cell } i \]
Computing FSMI

- MI contribution of a ray is a function of cell occupancy values:

Time Taken $\propto$ ray length
Computing FSMI

- Accumulate the MI contribution from all rays emanating from the cell:
Computing FSMI

- Accumulate the MI contribution from all rays emanating from the cell:
• Accumulate the MI contribution from all rays emanating from the cell:
Accumulate the MI contribution from all rays emanating from the cell:
Computing FSMI

- Accumulate the MI contribution from all rays emanating from the cell:
Computing FSMI

• Accumulate the MI contribution from all rays emanating from the cell:

\[ MI_{\text{location}} = \sum_{j=1}^{\text{num. rays}} MI_{\text{ray}_j} \]
• Accumulate the MI contribution from all rays emanating from the cell:

\[
MI_{\text{location}} = \sum_{j=1}^{\text{num. rays}} MI_{\text{ray}_j}
\]
Computing FSMI

• Accumulate the MI contribution from all rays emanating from the cell:

\[ \text{Time Taken} \propto (\text{ray length } \times \text{num. of rays}) \]
Computing FSMI

- MI value calculation at another cell:

Time Taken $\propto (\text{ray length } \times \text{num. of rays})$
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

- Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

- Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

- Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Repeat for every cell on the map!
Computing FSMI

• Now we want to compute the MI at all locations on the map!

Time Taken

\( \propto \)

(ray length \( \times \) num. of rays \( \times \) num. of cells)
Computing FSMI

• Now we want to compute the MI at all locations on the map!

<table>
<thead>
<tr>
<th>Time Taken</th>
<th>(ray length × num. of rays × num. of cells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>200 × 200</td>
</tr>
</tbody>
</table>
Computing FSMI

- Now we want to compute the MI at all locations on the map!

Time Taken

\[ \alpha \]

(ray length \times \text{num. of rays} \times \text{num. of cells})

\[
\begin{array}{ccc}
200 & 60 & 200 \times 200 \\
\end{array}
\]

\[ \sim 480 \text{ million operations} \]
Improvements in FCMI

- Consider the MI contributions from the following rays:
Improvements in FCMI

• Consider the MI contributions from the following rays:
Improvements in FCMI

• Consider the MI contributions from the following rays:

Redundant compute is performed.
Improvements in FCMI

- Consider the MI contributions from the following rays:
• Consider the MI contributions from the following rays:

A lot of redundant compute is performed.
Improvements in FCMI

• Define MI contributions from rays recursively:
Improvements in FCMI

- Define MI contributions from rays recursively:
Improvements in FCMI

- Define MI contributions from rays recursively:
Improvements in FCMI

- Define MI contributions from rays recursively:
Improve in FCMI

- Define MI contributions from rays recursively:
Improvements in FCMI

- Compute all MI contributions along this ray in one pass:

\[ \text{Time Taken} \propto \text{ray length} \]
Improvements in FCMI

• Compute all MI contributions along these rays:

\[ \text{Time Taken} \propto \text{num. of cells} \]
Improvements in FCMI

- Compute all MI contributions along these rays:

\[ \text{Time Taken} \propto \text{num. of cells} \]
Improvements in FCMI

• Compute all MI contributions along these rays:

Time Taken $\propto$ num. of cells
Improvements in FCMI

• Compute all MI contributions along these rays:

Time Taken $\propto$ num. of cells
Improvements in FCMI

• Compute all MI contributions along these rays:

\[
\text{Time Taken } \propto \text{ num. of cells}
\]
Improvements in FCMI

- Compute all MI contributions along these rays:

\[ \text{Time Taken} \propto \text{num. of cells} \]
Improvements in FCMI

- Compute all MI contributions along these rays:

\[ \text{Time Taken } \propto \text{num. of cells} \]
Improvements in FCMI

• Compute all MI contributions along these rays:

\[\text{Time Taken } \propto \text{ num. of cells}\]
Improvements in FCMI

• Total runtime of the algorithm:

\[
\text{Time Taken} \propto (\text{num. of rays} \times \text{num. of cells})
\]
Improvements in FCMI

• Total runtime of the algorithm:

\[
\text{Time Taken} \propto \frac{\text{ray length}}{(\text{num. of rays} \times \text{num. of cells})}
\]
Improvements in FCMI

• Total runtime of the algorithm:

\[
\text{Time Taken} \propto (\text{num. of rays} \times \text{num. of cells}) \\
60 \times 200 \times 200 \\
\sim 2.4 \text{ million operations}
\]
Tradeoffs in FCMI

• Compute complexity is lowered:
  – (480M operations v/s 2.4M operations)
Tradeoffs in FCMI

• Compute complexity is lowered:
  – (480M operations v/s 2.4M operations)

• Requires additional storage for partial MI values.
Tradeoffs in FCMI

• Compute complexity is lowered:
  – (480M operations v/s 2.4M operations)

• Requires additional storage for partial MI values.

• Recursive algorithm challenges parallel hardware design.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

An example sensor ray

Data dependencies for computation along the ray

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

• This recursive computation creates a data dependency.

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

An example sensor ray

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

**An example sensor ray**

**Data dependencies for computation along the ray**

**MI evaluation at a cell must finish before evaluating the next cell along the ray.**
Tradeoffs in FCMI

• This recursive computation creates a data dependency.

An example sensor ray

Data dependencies for computation along the ray

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

An example sensor ray

Data dependencies for computation along the ray

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Tradeoffs in FCMI

- This recursive computation creates a data dependency.

An example sensor ray

Data dependencies for computation along the ray

MI evaluation at a cell must finish before evaluating the next cell along the ray.
Parallelized FCMI

- This work exploits two types of parallelism:
  - Pipeline parallelism
  - Multi-core parallelism
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:
Parallelized FCMI: Multi-core

• Computations for distinct range measurement rays are independent.

• A multicore system can compute them in parallel.
  – Each core can compute MI along one ray:

\[ t = 1 \]
• Computations for distinct range measurement rays are independent.

• A multicore system can compute them in parallel.  
  – Each core can compute MI along one ray:

![Diagram showing parallelized FCMI]
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:

![Diagram showing parallel computation]

$\begin{align*}
\text{Core 1} & \quad \text{Green} \\
\text{Core 2} & \quad \text{Blue} \\
\text{Core 3} & \quad \text{Gray} \\
\text{Core 4} & \quad \text{Red} \\
t = 3
\end{align*}$
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:
Computations for distinct range measurement rays are independent.

A multicore system can compute them in parallel.

- Each core can compute MI along one ray:
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:

![Diagram showing parallelized FCMI with cores 1, 2, 3, and 4 computing rays at t = 7]
Parallelized FCMI: Multi-core

- Computations for distinct range measurement rays are independent.
- A multicore system can compute them in parallel.
  - Each core can compute MI along one ray:
Parallelized FCMI: Multi-core

High Memory Bandwidth is required for multi-core parallelism.
Parallelized FCMI: Multi-core

High Memory Bandwidth is required for multi-core parallelism.

Memory Banking!
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:

- We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:

- We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

- Instead of computing one cell MI update at a time:

- We use pipeline parallelism to compute multiple cells:
Parallelized FCMI: Pipeline

• Instead of computing one cell MI update at a time:

• We use pipeline parallelism to compute multiple cells:
Pipelining is not straightforward due to data dependencies.
Parallelized FCMI: Pipeline

Pipelining is not straightforward due to data dependencies.

Ray Interleaving!
Hardware Architecture Overview
Hardware Architecture Overview

Controller

Cell Coordinate Generation

Memory Subsystem

(Occupancy Grid Storage)

(Partial MI Storage)

MI Cores

Core 1

Core 2

Core 3

Core 16
Hardware Architecture Overview

- **Memory Subsystem**
  - Memory Access Pattern
  - Memory Banking

![Diagram showing Memory Subsystem, Core Connections, and Memory Subsystem Components]
Hardware Architecture Overview

- **Memory Subsystem**
  - Memory Access Pattern
  - Memory Banking

- **Mutual Information Core**
  - Pipelining
  - Ray Interleaving
  - Workload Balancing
  - Arithmetic Optimizations
Hardware Architecture Overview

- Memory Subsystem
  - Memory Access Pattern
  - Memory Banking
- Mutual Information Core
  - Pipelining
  - Ray Interleaving
  - Workload Balancing
  - Arithmetic Optimizations

- Putting it all together: top-level architecture!
Memory Subsystem

- Memory is implemented as SRAMs.
  - A standard SRAM has two access ports.
  - This means only two cores active at a time, at best.

**Bottleneck increases system latency.**
Memory Subsystem

- Memory is implemented as SRAMs.
  - A standard SRAM has two access ports.
  - This means only two cores active at a time, at best.

Bottleneck increases system latency.

- Split the storage across multiple SRAMs or “banks”!
  - Need to determine optimal partitioning scheme.
Memory Subsystem

• The memory access pattern is fixed and deterministic:
  – $n$ consecutive cells are always accessed together by $n$ cores.
Memory Subsystem

- The memory access pattern is fixed and deterministic:
  - $n$ consecutive cells are always accessed together by $n$ cores.
  - Example with $n = 4$ cores:
The memory access pattern is fixed and deterministic:

- $n$ consecutive cells are always accessed together by $n$ cores.
- Example with $n = 4$ cores:
Memory Subsystem

• Idea: use a diagonal banking pattern.
  – Cells along a diagonal are in the same bank.
  – Example with $m = 4$:
Memory Subsystem

• Any $n = 4$ consecutive cells guaranteed to lie in distinct banks:
• Any $n = 4$ consecutive cells guaranteed to lie in distinct banks:
Any $n = 4$ consecutive cells guaranteed to lie in distinct banks:
Memory Subsystem

• Any $n = 4$ consecutive cells guaranteed to lie in distinct banks:
Memory Subsystem

- Each core accesses distinct banks.
  - Results in zero memory access related stalls.
Memory Subsystem

- Each core accesses distinct banks.
  - Results in zero memory access related stalls.

- Diagonal banking is easy to implement in hardware.
Memory Subsystem

• Each core accesses distinct banks.
  – Results in zero memory access related stalls.

• Diagonal banking is easy to implement in hardware.

• Advantage over FSMI: no arbiter needed.
• Evaluation of MI at each core can be performed as follows:
MI Core: Pipelining

- Evaluation of MI at each core can be performed as follows:

- Each evaluation can be broken up into smaller operations:

```
+----------------+----------------+----------------+
| Clock Cycle    | Clock Cycle    | Clock Cycle    |
| Cell 1         | Cell 2         | Cell 3         |
| Time           | Time           | Time           |
```

```
+----------------+----------------+----------------+----------------+----------------+
| Cell 1         | Cell 1         | Cell 1         | Cell 1         | Cell 2         |
| Cell 2         | Cell 2         | Cell 2         | Cell 2         | Cell 2         |
| Cell 3         | Cell 3         |                |                |                |
| Time           | Time           | Time           | Time           | Time           |
```
**MI Core: Pipelining**

- Each smaller operation can be implemented on separate hardware, and parallelized:

<table>
<thead>
<tr>
<th></th>
<th>Cell 1</th>
<th>Cell 1</th>
<th>Cell 1</th>
<th>Cell 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cell 2</td>
<td>Cell 2</td>
<td>Cell 2</td>
<td>Cell 2</td>
</tr>
<tr>
<td></td>
<td>Cell 3</td>
<td>Cell 3</td>
<td>Cell 3</td>
<td>Cell 3</td>
</tr>
</tbody>
</table>

Clock Cycle  \hspace{1cm} Time

Pipelining leads to faster clock cycle, therefore **lower latency**.
Recall that MI computation along a ray has data dependencies.

Cell 2 cannot begin before Cell 1 computation finishes!
MI Core: Ray Interleaving

- Solution: Compute several independent rays on one core!

Ray Interleaving preserves the fast clock cycle, while avoiding data dependency conflict!
MI Core: Workload Balancing

- Execution time for each ray is proportional to its length:

  - Max length rays
  - Complementary rays that add up to max length
• Execution time for each ray is proportional to its length:

- Max length rays
- Complementary rays that add up to max length
• Execution time for each ray is proportional to its length:

Balance core workloads by computing complementary rays sequentially!
MI Core: Arithmetic Optimizations

• Logarithms and reciprocal of the cell occupancy probability:
  – Use pre-computed look-up table
MI Core: Arithmetic Optimizations

• Logarithms and reciprocal of the cell occupancy probability:
  – Use pre-computed look-up table

• Exponential for a larger input space:
  – Use a piecewise linear approximation
MI Core: Arithmetic Optimizations

• Logarithms and reciprocal of the cell occupancy probability:
  – Use pre-computed look-up table

• Exponential for a larger input space:
  – Use a piecewise linear approximation

• High precision is not a requirement:
  – Use fixed point instead of floating point
Top-level Architecture
Top-level Architecture

Controller

ATU 1
ATU 2
ATU 3
ATU 16

Crossbar Interconnect

Occupancy Grid Bank 1
Mutual Information Bank 1

Occupancy Grid Bank 2
Mutual Information Bank 2

Occupancy Grid Bank 3
Mutual Information Bank 3

Memory Subsystem

Core 1
Core 2
Core 3
Core 16

MI Cores
Top-level Architecture
Top-level Architecture

Controller

ATU 1

ATU 2

ATU 3

ATU 16

Crossbar Interconnect

Occupancy Grid Bank 1
Mutual Information Bank 1

Occupancy Grid Bank 2
Mutual Information Bank 2

Occupancy Grid Bank 3
Mutual Information Bank 3

ATU 1

ATU 2

ATU 3

ATU 16

Crossbar Interconnect

Memory Subsystem

Core 1

Core 2

Core 3

Core 16

MI Cores
Hardware Evaluation Results
Evaluation: Accuracy

- We compare the results from our hardware to a reference software implementation:

Occupancy Grid Map

(201x201 cells)
Evaluation: Accuracy

- We compare the results from our hardware to a reference software implementation:

![Images of Occupancy Grid Map, Reference MI Map, and Generated MI Map](image)
Evaluation: Accuracy

- We compare the results from our hardware to a reference software implementation:
We compare the results from our hardware to a reference software implementation:

Our hardware is sufficiently accurate for exploration!
Evaluation: Latency

- We evaluate the impact of our implementation features on the latency:

![Graph showing latency vs. number of cores.](image-url)
We evaluate the impact of our implementation features on the latency:

![Graph showing latency vs. number of cores](image-url)
Evaluation: Latency

- We evaluate the impact of our implementation features on the latency:
We evaluate the impact of our implementation features on the latency:
Evaluation: Latency

We evaluate the impact of our implementation features on the latency:
Evaluation: Latency

- We evaluate the impact of our implementation features on the latency:

Latency is 1.55 ms i.e. much faster than real-time.
We also compare the latency of our architecture against a variety of platforms:

Latency is two orders of magnitude lower than the previous state of the art!
Evaluation: Energy Consumption

- We measure the energy consumed per MI map update, against the number of cores in the architecture:

Only 1.7 mJ of energy is consumed per MI update.
Evaluation: Energy Consumption

- We also compare the energy consumed per MI map update, against a variety of platforms:

![Energy Consumption Chart]

Energy consumption is **two orders of magnitude lower** than previous state of the art!
Evaluation: Impact on Exploration

- We run an exploration experiment, simulating MI computation on different platforms:

  ![Graph showing entropy decrease over trajectory length for different platforms.]

  More frequent MI updates results in more efficient exploration!
Evaluation: Impact on Exploration

- We also measure the drop in map entropy against energy consumed on MI compute:

Three orders of magnitude less energy is spent on MI compute for a given drop in map entropy.
Key Takeaways

• Fast MI computation is key to efficient exploration!
  – Infeasible on current robots due to intensive compute.
  – Hardware design is challenged by the recursive algorithm
Key Takeaways

• Fast MI computation is key to efficient exploration!
  – Infeasible on current robots due to intensive compute.
  – Hardware design is challenged by the recursive algorithm

• We introduce a hardware accelerator for FCMI with:
  – High bandwidth memory subsystem
  – Energy efficient MI compute cores
  – Uninterrupted pipeline to accelerate the recursive algorithm
Key Takeaways

• Fast MI computation is key to efficient exploration!
  – Infeasible on current robots due to intensive compute.
  – Hardware design is challenged by the recursive algorithm

• We introduce a hardware accelerator for FCMI with:
  – High bandwidth memory subsystem
  – Energy efficient MI compute cores
  – Uninterrupted pipeline to accelerate the recursive algorithm

• Compared to the previous state of the art, our accelerator:
  – Runs more than 100x faster.
  – Consumes more than 180x less energy.
Key Takeaways

• Fast MI computation is key to efficient exploration!
  – Infeasible on current robots due to intensive compute.
  – Hardware design is challenged by the recursive algorithm

• We introduce a hardware accelerator for FCMI with:
  – High bandwidth memory subsystem
  – Energy efficient MI compute cores
  – Uninterrupted pipeline to accelerate the recursive algorithm

• Compared to the previous state of the art, our accelerator:
  – Runs more than 100x faster.
  – Consumes more than 180x less energy.

We enable real-time MI compute for the first time!