# Legion Profiler

Seema Mirchandaney (SLAC), Elliott Slaughter (SLAC), Wei Wu (NVIDIA)

# Agenda

- Legion Profiler
- Realm Profiler
- Future Works

# Overview of Legion Profiler (Legion Prof)

# Measurement and Logging

## Subtitle

- Seamlessly done by Legion/Realm
  - Legion issues profiling request to Realm
  - Realm records measurements during execution
  - profiling tasks are launched to collect data when measurements are done
- Task-based profiler
  - Task
  - Copy
    - Intra-node, inter-node, host-device, device-device
  - Instance
  - Runtime-internal operation
    - Mapper Call, Runtime Call, Meta Task, Profiling Task, …
- Format of output data
  - ZLIB for performance
  - Plain text

**Steps:**

1. Compile the application with **DEBUG=0**
2. Profile the application

Binary format:

./app -lg:prof <N> -lg:prof_logfile prof_%.gz

Text format:

./app -lg:prof <N> -logfile prof_%.txt -lg:serializer ascii

# Data Processing

## Subtitle

- Two implementations
  - Python
  - Rust
- Out-of-order processing
  - A task/copy/… could  be recorded before its predecessors
  - Lazy recording of memories/processors affinities
- Rust implementation
  - Now at feature parity with Python
    - Bitwise identical output in CI
  - 20x faster (or more) than Python
    - 2x less memory
    - **Easier to fit large profiles**
  - Use same visualization frontend as Python

**Python:**

$LG_RT_DIR/../tools/legion_prof.py prof_*.gz

**Rust**:

1. Install Rust:
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

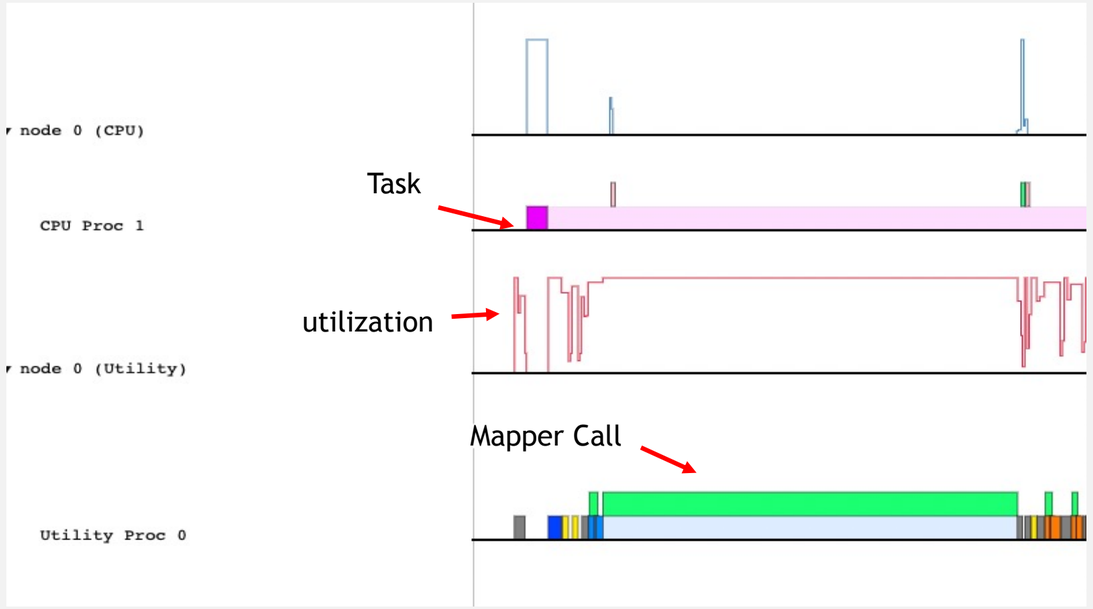
2. Compile and install Legion Prof:
cargo install --path legion/tools/legion_prof_rs

3. Run Legion Prof (support most of the same flags as Python):
legion_prof prof_*.gz

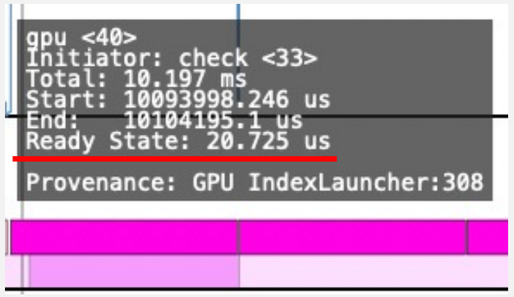
# Visualization

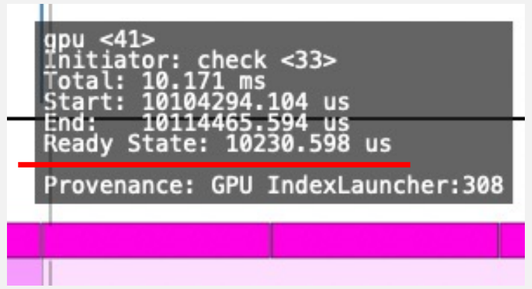
## Overview and Processor

- D3 JavaScript library with html, svg and css
- Y-axis: Processors, Memories and Channels
- X-axis: Timeline of Tasks, Operations and Instances
- Processor
  - CPU/GPU/Python/…
  - Utilization
- Task (User Task, Prof Task)
  - Variant, initiator, timeline, …
  - 4 timelines: create, ready, start, end
  - waiting, ready: sub-tasks
- Operations (Runtime): Utility Proc
  - Mapper Call
  - Runtime Call
  - Meta Task

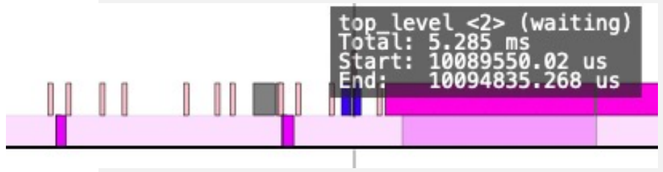



Processor

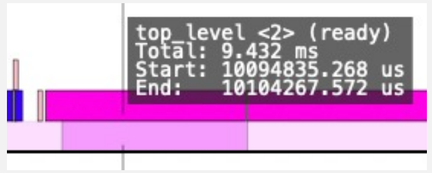



Ready State Delay




Wait for sub-tasks

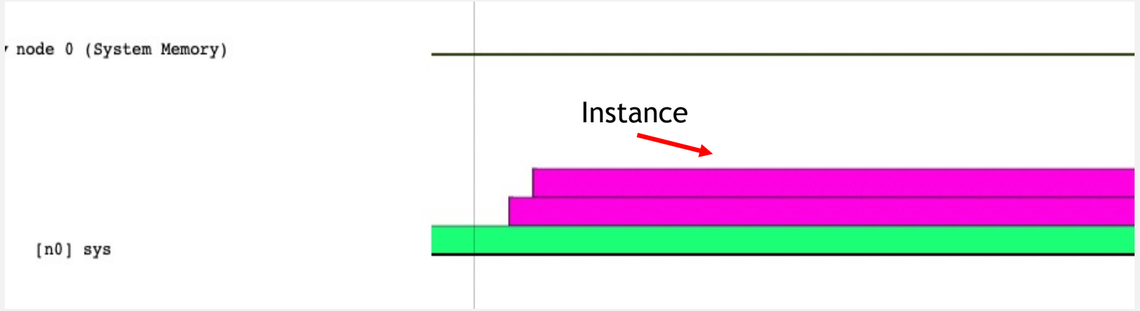



Wait for resources

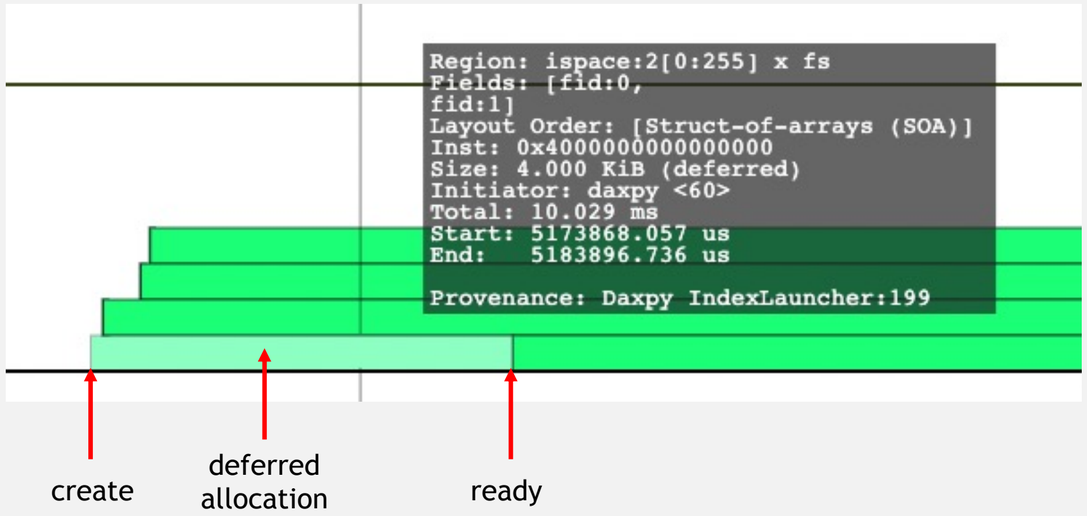
# Visualization (con't)

## Memory

- Memory
  - System memory
  - External system memory (attach operation)
  - Frame buffer memory
  - Zero copy memory
  - …
- Instance
  - Layout constraints
    - Row/column major
    - AOS, SOA
  - Regions
    - Index space, field space, fields, sparsity
  - Size
  - Deferred allocation (shaded)
    - Cost from create to ready




Memory

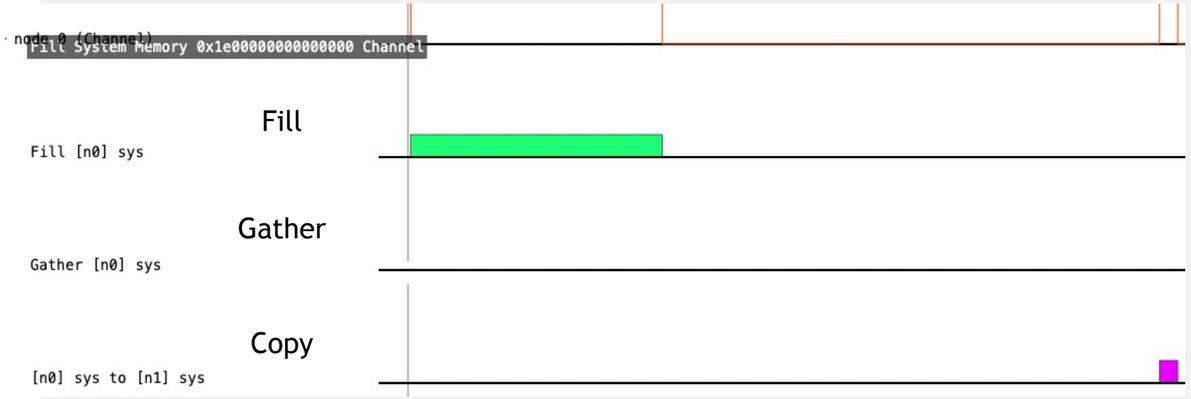



Instance

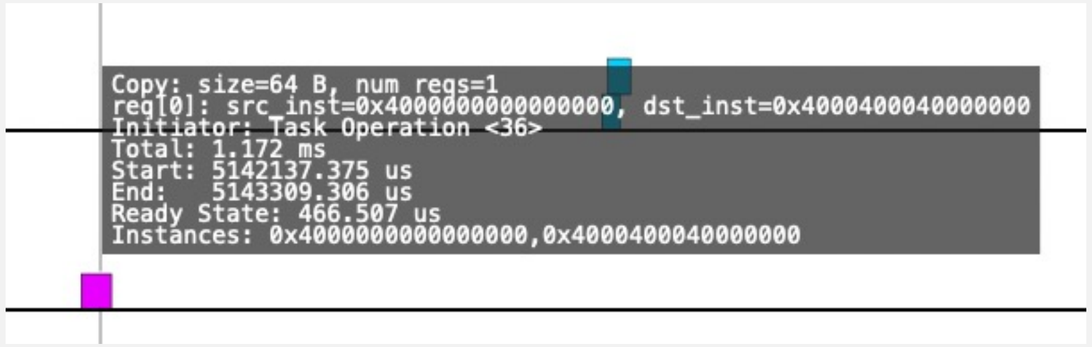
# Visualization (con't)

## Channel

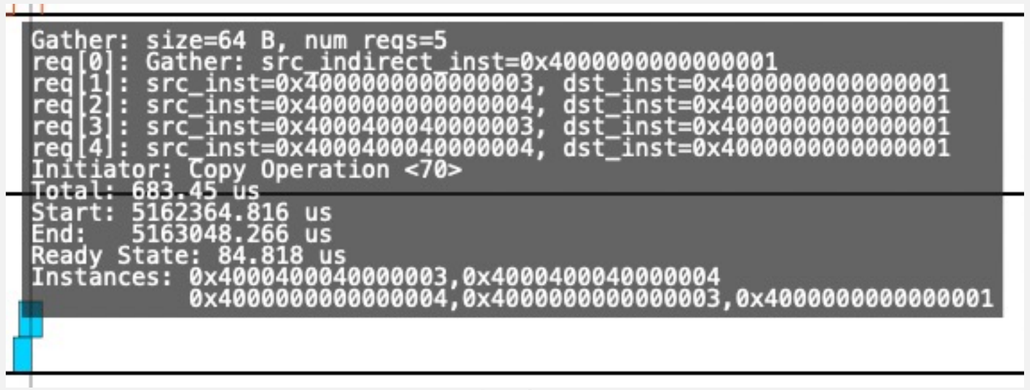
- Channel
  - Copy: src_dst
  - Fill: dst
  - Indirect Copy: Gather: dst, Scatter: src
  - Dependent Partition
- Copy/Fill
  - Size
  - Info (src and dst instances) of each copy request
  - Initiator
  - Ready State: cost between ready and start
- Indirect Copy (Gather/Scatter)
  - Size
  - Meta data of indirect copy
  - Info of each copy request
  - Initiator
  - Ready State




Channel




Copy

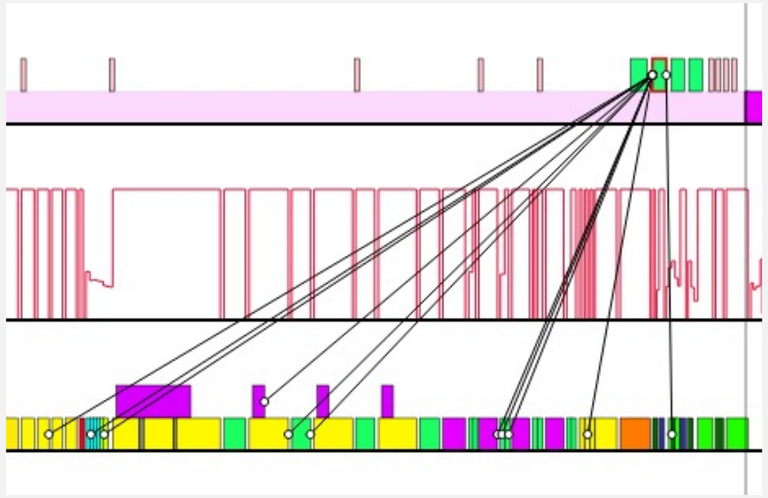



Gather

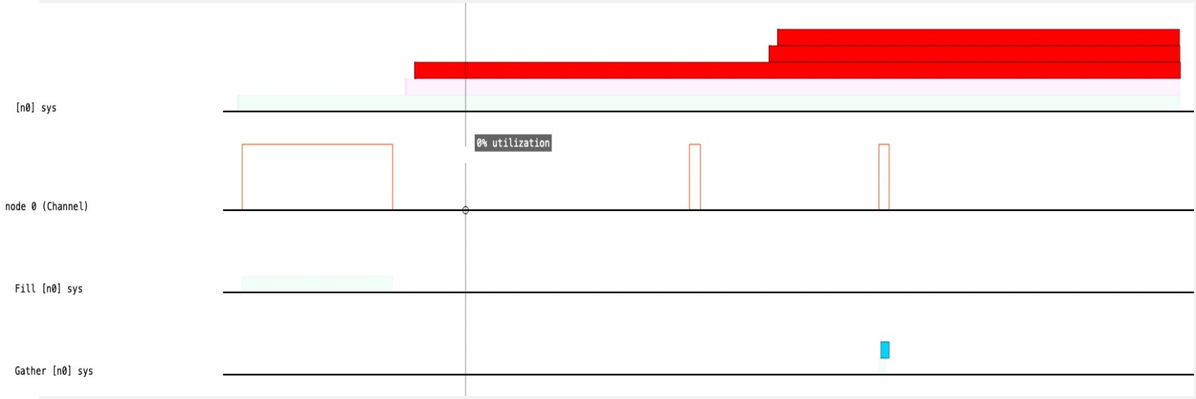
# Visualization (con’t)

## Advanced Features

- Task Dependencies
  - Interpret Legion Spy data to draw dependencies
  - Run with Legion Spy (-lg:spy -logfile spy_%.log)
- Associate Instances with Copy/Fill/Task
  - Highlight the instances used by a copy, fill or task
- Option Menu
  - Select/view a subset of nodes and memories/processors



Task Dependencies

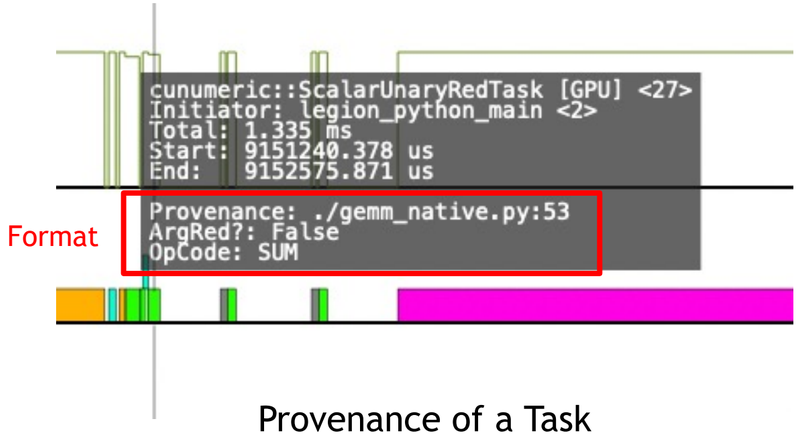



Instances of a Gather

# Visualization (con't)

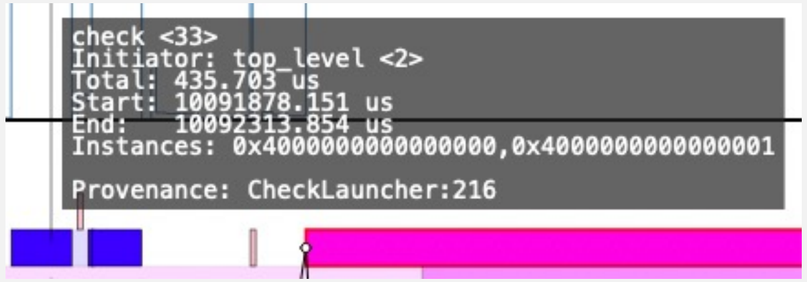
## Advanced Features

- Provenance
  - How to map a task/copy/… to lines in code ?
  - Supported by Legion APIs on Launchers, Partitions, …
  - Format
    - (human readable string)$(key1),(value1)|(key2),(value2)|...
  - Provenance of Task
    - Provenance string from user
  - Provenance of Other Operations
    - Provenance string of initiator

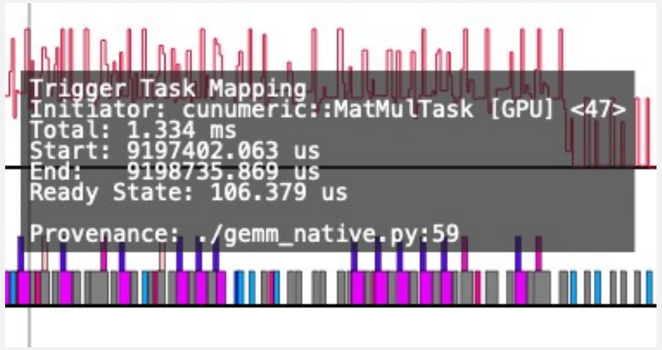



Provenance of a Task

```
TaskLauncher check_launcher(CHECK_TASK_ID,
    TaskArgument(&alpha, sizeof(alpha)));
std::string check_launcher_prov =
    "CheckLauncher:" + std::to_string(__LINE__);
check_launcher.provenance = check_launcher_prov;
```




Example of Using Provenance

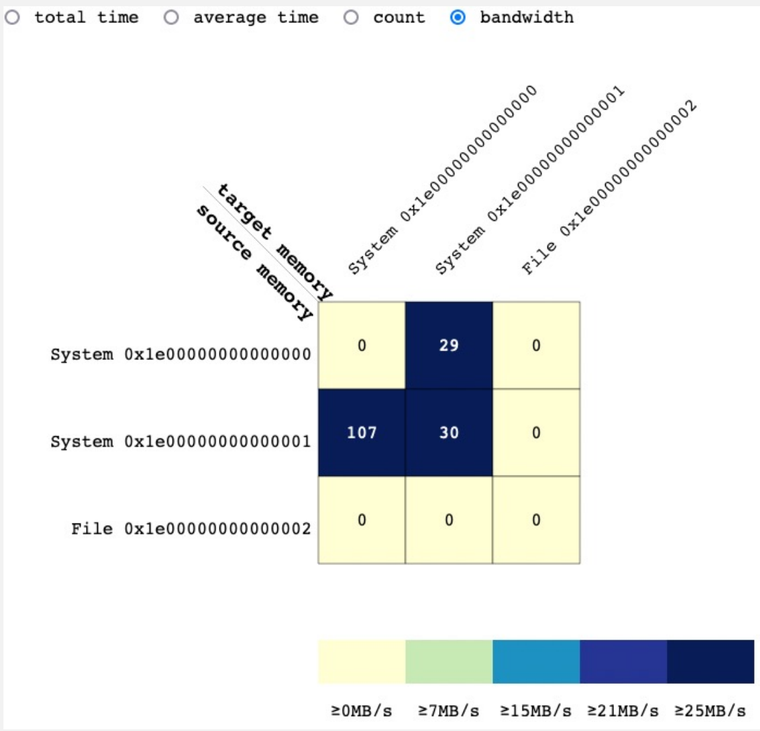



Provenance of a Runtime Operation

# Visualization (con't)

## Advanced Features

- Copy Matrix
  - Display the latency and bandwidth between a pair of memories
  - Enabled by -C
- Message Warning
  - Notify slow data movements
  - --message-threshold <N> –-message-percentage <P>
    - Warn if p% of the data movements take more than N microsecond
- Statistics
  - Statistic overview for each Processor, Memory, Channel, Task Variant, MetaTask Variant and Mapper
  - Enabled by -s




Copy Matrix

```
CPU Processor 0x1d00000000000001
    Total time: 116640 us
    Active time: 48137 us (41.270%)
    Application time: 115594 us (99.104%)
    Meta time: 1045 us (0.896%)
    Mapper time: 0 us (0.000%)
```

Statistics of Processor

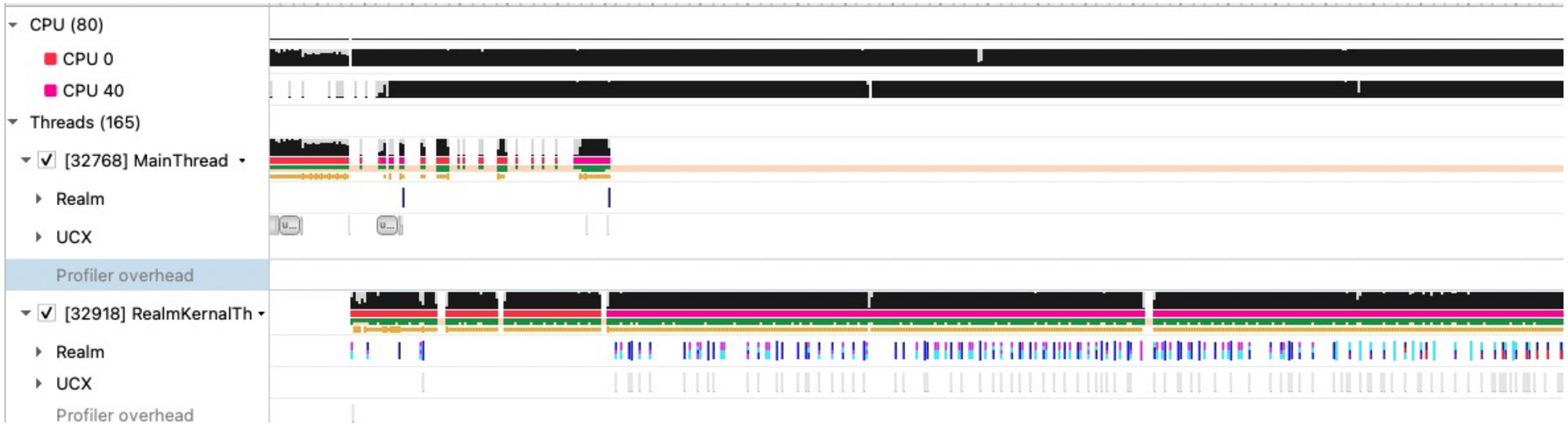
```
daxpy
     Total Invocations: 4
     Total Time: 702.42 us
     Average Time: 175.60 us
     Maximum Time: 200.69 us (7.083 sig)
     Minimum Time: 162.70 us (-3.643 sig)
```

Statistics of Task

# Realm Profiler

## NVTX

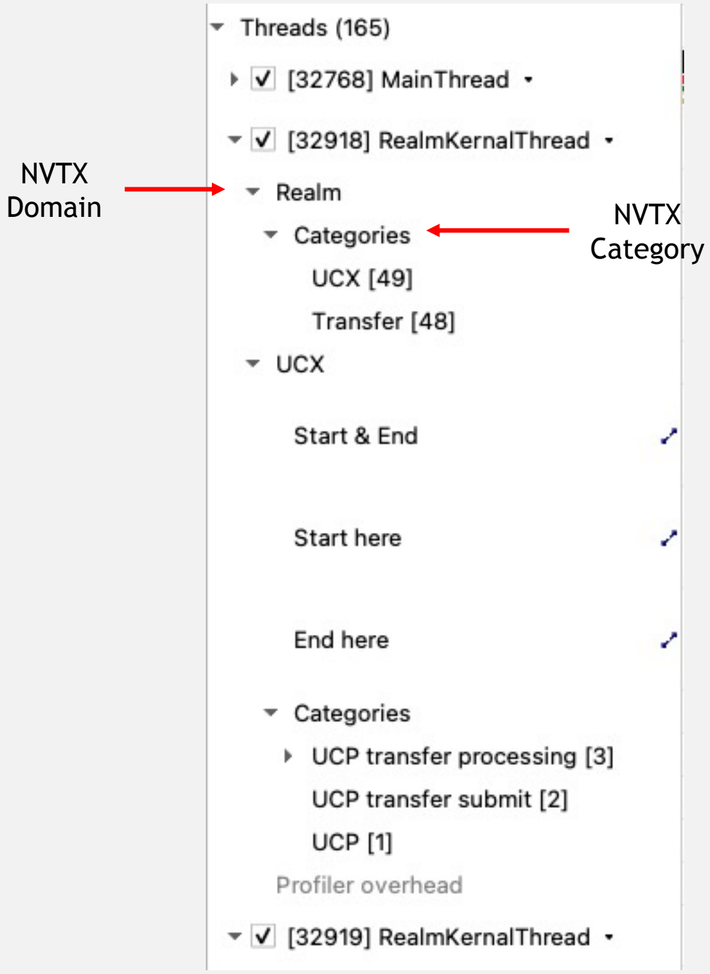
- Goal
  - Legion Profiler is coarse grind (task-based)
  - Investigate the performance of operations within Realm
- NVTX
  - Better visualization tool (Nsight system)
  - Support variety of libraries: CUDA, CUDNN, MPI, UCX, …
  - NVTX-T
  - No multi-node support



# Realm Profiler

## NVTX

- Thread-level profiler
- NVTX Category
- NVTX Domain
  - Higher level category
  - Usually per library




Y-axis of NVTX Trace

# Realm Profiler

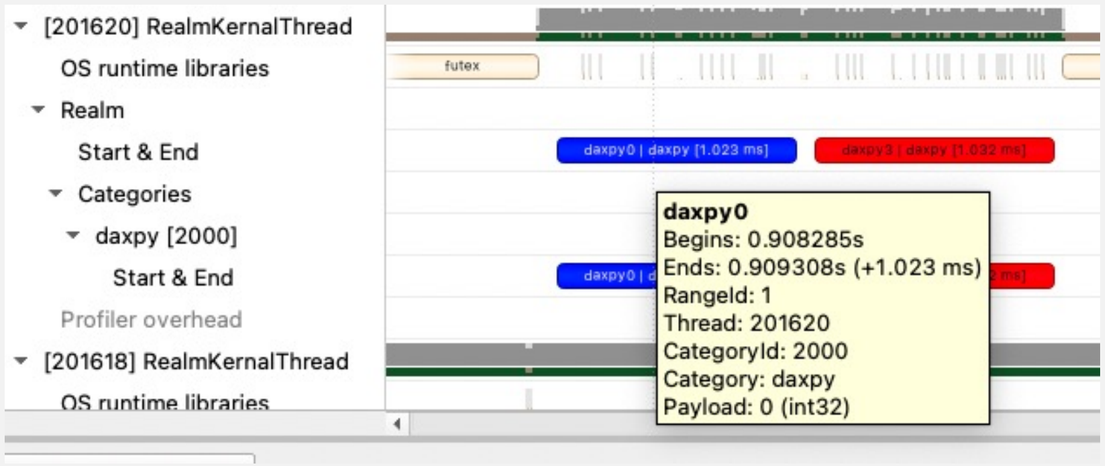
## NVTX Realm Module

- NVTX Wrapper
  - Allow people to instrument code easily
  - More efficient than original NVTX
- Build NVTX Module
  - CMake: Legion_USE_NVTX=ON
  - Makefile: USE_NVTX=1
- Thread Local (Thread-safe) Category
  - Predefined category (empty)
    - amsg, bgwork, cuda, hip, gasnet1, gasnetex, …
    - -ll:nvtx_modules (all/amsg/bgwork/…)
  - User defined category
    - For applications
- Thread Specific Category
  - Processor: visualize tasks

```
Realm::create_user_category(
  "daxpy", Realm::nvtx_color::red);

return Runtime::start(argc, argv);
```

Create User Category

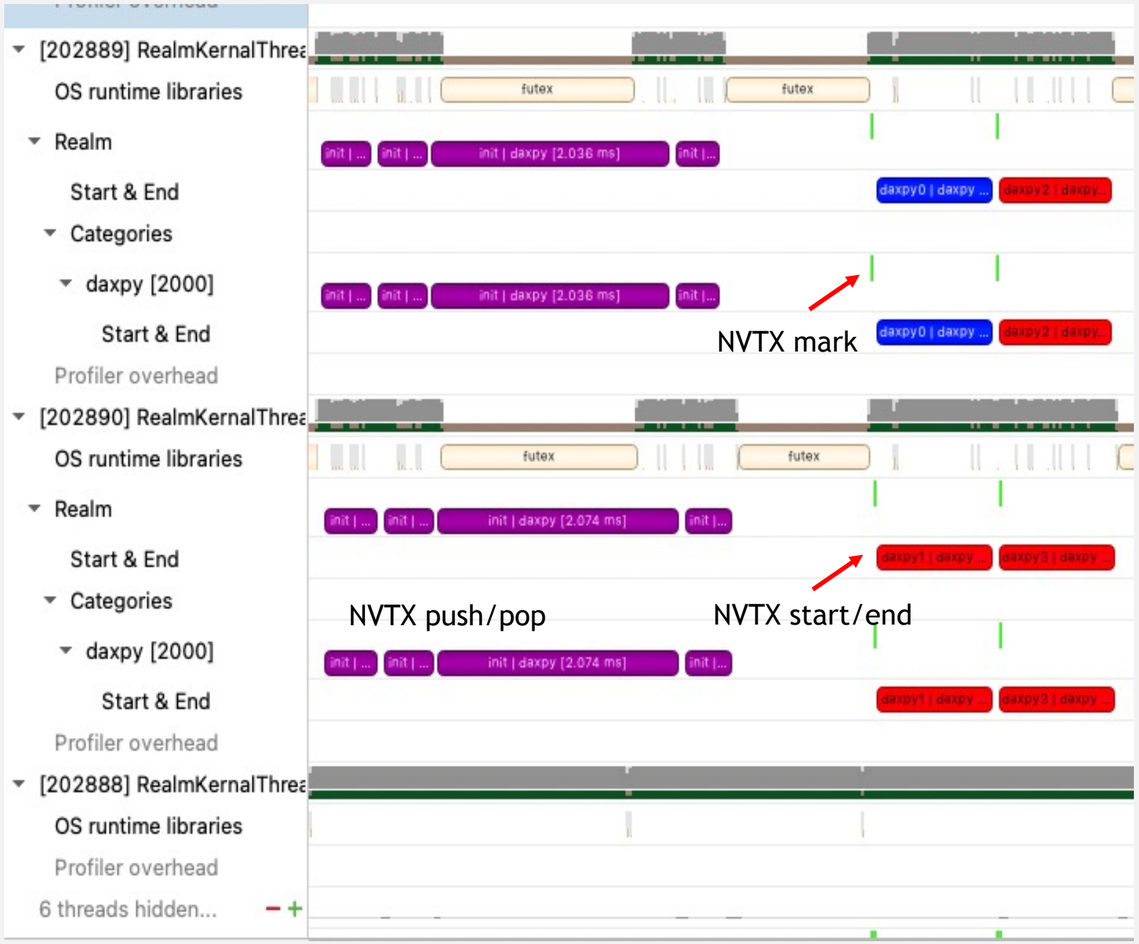



User Category Visualization

# Realm Profiler

## API of NVTX Realm Module

- Start/End
  - nvtx_range_start(category_name, message, color, …)
  - nvtx_range_end()
- Stack-based
  - nvtx_range_push(category_name, message, color, …)
  - nvtx_range_pop()
  - nvtxScopedRange{category_name, message, color, …}
- Marker
  - nvtx_mark(category_name, message, color, …)




NVTX Visualization

# Future Works

- Legion Profiler
  - Visualize Index Task
  - Show movement of future values
  - Support ScatterGather Channel
  - Visualize eager pool
  - Show all OpenMP worker thread activity
  - New visualization tools
- Realm Profiler
  - Associate realm profiling trace with Legion Profiler
  - Query Realm metrics (Currently measured by NVTX)
  - Collect network congestion information
    - GASNet-provided active message statistics, IB counters, NIC packet counters, …
- Automatic Analysis of Profiling Data
  - Collect data from different sources including Legion Prof, Legion Spy, Realm Prof, …
  - Warn people possible performance issues.


Q
A