

Snoopie: A Multi-GPU Communication Profiler and Visualizer

Mohammad Kefah Taha Issa*
Koç University
Istanbul, Türkiye
missa18@ku.edu.tr

Muhammad Aditya Sasongko*
Koç University
Istanbul, Türkiye
msasongko@ku.edu.tr

Ilyas Turimbetov
Koç University
Istanbul, Türkiye
iturimbetov18@ku.edu.tr

Javid Baydamirli
Koç University
Istanbul, Türkiye
jbaydamirli21@ku.edu.tr

Doğan Sağbili
Koç University
Istanbul, Türkiye
dsagbili17@ku.edu.tr

Didem Unat
Koç University
Istanbul, Türkiye
dunat@ku.edu.tr

ABSTRACT

With data movement becoming one of the most expensive bottlenecks in computing, the need for profiling tools to analyze communication becomes crucial for effectively scaling multi-GPU applications. While existing profiling tools including first-party software by GPU vendors are robust and excel at capturing compute operations within a single GPU, support for monitoring GPU-GPU data transfers and calls issued by communication libraries is currently inadequate. To fill these gaps, we introduce Snoopie, an instrumentation-based multi-GPU communication profiling tool built on NVBit, capable of tracking peer-to-peer transfers and GPU-centric communication library calls. To increase programmer productivity, Snoopie can attribute data movement to the source code line and the data objects involved. It comes with multiple visualization modes at varying granularities, from a coarse view of the data movement in the system as a whole to specific instructions and addresses. Our case studies demonstrate Snoopie's effectiveness in monitoring data movement, locating performance bugs in applications, and understanding concrete data transfers abstracted beneath communication libraries. The tool is publicly available at https://github.com/ParCoreLab/snoopie.

CCS CONCEPTS

• **General and reference** → **Performance**; **Measurement**; **Metrics**; **Evaluation**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

ACM Reference Format:
Mohammad Kefah Taha Issa, Muhammad Aditya Sasongko, Ilyas Turimbetov, Javid Baydamirli, Doğan Sağbili, and Didem Unat. 2024. Snoopie: A Multi-GPU Communication Profiler and Visualizer. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 4–7, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656597

*These authors contributed equally

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICS '24, June 4–7, 2024, Kyoto, Japan

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656597

1 INTRODUCTION

Application development and performance scaling on multiple GPUs is nontrivial and often require continuous profiling and debugging. GPU communication mechanisms are also diverse, with prior methods requiring data transfers to pass through the host CPU. However, advancements in direct peer-to-peer communication technology such as NVLink and GPUDirect [24] have enabled GPUs to communicate directly with their peers during data transfers, bypassing the intermediate host buffers. These transfers can be initiated by either the host or device and can involve direct access to remote memory through loads or stores.

However, despite the advancements in data movement technologies, existing multi-GPU profiling tools have lagged behind, offering limited information about data movement. Nsight Systems [48], a performance analysis tool developed and maintained by Nvidia, *vaguely* indicates the presence of device-initiated direct memory accesses as traffic between NVLink-connected GPUs but does not provide further information to end-users such as devices engaged in such communication. HPCToolKit [1], which is a suite of performance analysis tools for CPUs and GPUs, can track the amount of data moved among devices but it is restricted to cases where communication calls are initiated by the host. Similarly, the majority of other profiling tools offer limited or no insights into data movement caused by communication libraries like NVSHMEM [44] and NCCL [53]. The latter is utilized by popular deep learning frameworks, including PyTorch [7] and TensorFlow [10], to facilitate multi-GPU acceleration.

Furthermore, most tools [8, 31, 48, 64] including those that do support said communication operations lack source code and object attribution, providing an incomplete picture in their visualization. Attributing communication to the source code and involved program objects can greatly aid programmers in identifying scalability bottlenecks and data movement-related bugs. As a result, a diagnostic tool to address the aforementioned gaps is needed for effective development and debugging on multi-GPU systems.

This paper presents Snoopie, a complementary profiler that is designed specifically to address the shortcomings in existing mainstream tools, with the issues mentioned above in mind. Snoopie provides finer-grained information on C/C++ and Python-based GPU applications by detecting host-initiated transfers and peer-to-peer communication including device-initiated direct accesses, while providing support for commonly used communication libraries. It can identify the data objects accessed by the inter-GPU

Profiler	cudaMemcpy and its family	P2P Direct Access	Communication Library Support	Src Code Line Attribution	Data Object Attribution	Visualization
Nsight Systems [48]	✓	Limited	Limited	-	-	timeline
HPCToolkit [1]	✓	-	-	✓	-	code-centric
ComScribe[8]	✓	✓	only NCCL	-	-	comm matrix
EZTrace[64]	✓	-	-	-	-	timeline and comm matrix
Extrae[12]	✓	-	-	-	✓	timeline
Score-P[31]	✓	-	-	-	-	timeline
Snoopie (ours)	✓	✓	NCCL and NVSHMEM	✓	✓	multi-view, code-data-centric

Table 1: Comparing communication profiling tools for multi-GPU systems

memory operations and locate the source code lines where communication occurs. Lastly, it is equipped with a multi-view interactive visualization module that exposes various levels of granularity for users to obtain a broad overview of the system or a detailed view of a specific object or device. In short, our main contributions are as follows:

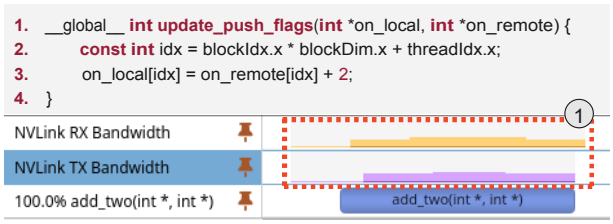
- An open-source profiling and analysis tool that captures inter-GPU data movement with support for NCCL and NVSHMEM communication libraries
- Support for communication association with the involved data objects and source code origin to ease analysis and debugging
- User-friendly visualization tool that presents GPU communication in various levels of granularity
- Methods to reduce profiling overhead to 2×-96× by on-device filtering, compression, and sampling
- Diverse set of case studies in C/C++ and Python that covers BFS, 2D Stencil, NCCL bandwidth test, and a Deep Learning CosmoFlow [36] model in PyTorch. to demonstrate the different capabilities of the tool.

While Snoopie is not intended to replace established profiling tools such as Nsight Systems, it complements them by filling a significant gap in the GPU tool portfolio. With a recent shift towards greater GPU autonomy, where devices directly manage communication [13, 23, 33], and the widespread use of communication libraries like NCCL and NVSHMEM, Snoopie becomes an important addition to the GPU toolset. It aids both application users and developers in effectively pinpointing communication bottlenecks and optimizing their code to achieve better performance. Lastly, although Snoopie's current implementation targets NVIDIA GPUs, many of our contributions are architecture-agnostic and can be adapted to other GPU vendors.

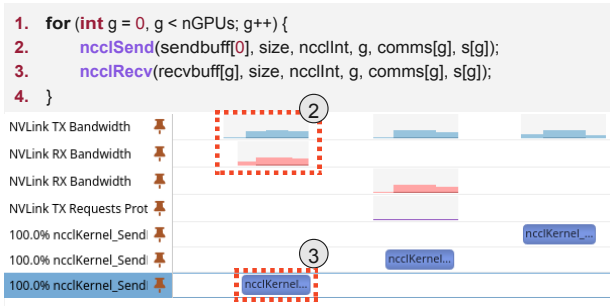
2 MOTIVATION: IDENTIFYING GAPS

Several profiling tools have been created for GPUs, with the main emphasis on examining the performance of computational kernels, with varying degrees of support for profiling communication calls. Table 1 lists some of these tools and their support for various aspects of communication profiling. However, we have identified certain gaps in their capabilities, including:

- Gap 1: Detecting different types of GPU-GPU communication including peer-to-peer (P2P) direct accesses to remote GPU memory
- Gap 2: Tracking data movement induced by communication libraries such as NVSHMEM and NCCL



(a) How P2P direct access appears in Nsight Systems



(b) How NCCL Send/Receive appears in Nsight Systems

Figure 1: Communication profiling in Nsight System. Figures are available under CC-BY [28].

- Gap 3: Ability to associate communication with the involved data objects and source code lines
- Gap 4: Granular visualization support for data movement for analysis and debugging

Gap 1: Detecting different types of GPU-GPU communication. CUDA provides several communication methods to facilitate data transfers among GPUs. The standard host-side API for data movement from one device to another is the `cudaMemcpy` family of calls, which are tracked by most existing profilers with the help of the CUDA runtime API. The same API allows peer-to-peer communication initiated by the host through `cudaMemcpyPeer` which utilizes DMA engines to perform data movement. Such communication can also be device-initiated when shared address space is available, where devices can directly load/store from/to another GPU's memory without involving the host [58]. This form of direct communication has gained prominence as more GPUs per node can be tightly connected via high-speed interconnects, facilitating low-latency access to peer GPU memories. P2P transfers also enable persistent kernels to communicate without termination, resulting

in fewer kernel launches, synchronization, and reduced networking overheads [6, 23, 24, 27, 33, 54, 55, 58].

Despite their importance, existing profiling tools currently lack support for device-initiated transfers, and only monitor host-initiated peer transfers. For example, NVIDIA's Nsight Systems [48] does not provide information on the effective addresses accessed by device-initiated remote memory operations or the devices that are the targets of the operations. Instead, its output as shown at ① in Figure 1a only *indicates* the occurrence of P2P accesses through a bump in NVLink traffic for a remote read. HPCToolKit [1], another commonly used HPC profiling tool, does not provide any information that could indicate the occurrence of P2P direct accesses.

Gap 2: Communication Library Support. NVIDIA provides two first-party communication libraries for multi-GPU applications: NVIDIA Collective Communications Library (NCCL) and NVSHMEM. NCCL supports both collective communication and point-to-point send/receive primitives for multi-GPU and multi-node systems. It has seen wide adoption in popular deep learning frameworks such as MxNet [18], PyTorch [7], and TensorFlow [10], which use it to accelerate distributed training. NVSHMEM extends the OpenSHMEM specification [57] to support NVIDIA GPUs and utilizes a global address space accessible with a fine-grained GPU-side API. [44, 58]. While Nsight Systems recognizes both NCCL and NVSHMEM calls, it only provides timeline information. Figure 1b highlights the basic profiling information gathered by Nsight Systems at points ② and ③ - showing only the timespan of NCCL calls, with no further information about the underlying properties of the operations.

Gap 3: Code Line and Object Attribution. Associating profiled communication operations with their source code origin and the involved data objects can aid programmers in identifying communication-related bugs and performance bottlenecks. Among the profiling tools listed in Table 1, we only identified two that provide this information: Extrae [12] and HPCToolkit. While Extrae can attribute data movement to data objects, this feature is limited to CPUs and there is no attribution for source code lines. HPCToolkit, on the other hand, is able to attribute communication to source code lines but lacks support for objects. Moreover, to the best of our knowledge, there are no tools that support code line and data object attributions in multi-GPU Python programs. Given the widespread use of Python in HPC and AI, such tooling becomes essential for GPU profiling in communication-heavy applications with complex codebases.

Gap 4: Data Movement Visualization. Current tools are generally multipurpose, and as a result, their visualization may not be specifically tailored to highlight data movement. Illustrated in Figure 1, most tools, including Nsight Systems, predominantly provide an overview of the execution time of profiled applications and offer, arguably, less informative representations of data transfers. Eztrace[64] and ComScibe[8] present a coarse-grained view of data movement in the form of a communication matrix. Having finer-grained information would assist users in understanding multi-GPU communication patterns, object access behaviors, and locating bottlenecks in their applications.

In summary, existing GPU profiling tools have limitations in offering a comprehensive understanding of inter-device communication in multi-GPU applications. Snoopie aims to fill the aforementioned gaps and provide an alternative perspective on communication analysis for end-users.

3 SNOOPIE WITH MULTI-VIEW VISUALIZER

Snoopie is a binary instrumentation tool that provides a data-centric and user-friendly way to monitor and analyze device-to-device communication. By tracking the source and destination devices of memory operations, the effective addresses, the amount of data transferred, the source code lines, and the accessed data objects, it provides a comprehensive overview of GPU communication.

Snoopie provides multiple intuitively understandable ways of visually representing the data transfers. The visualization module presents the logged information in an interactive manner, enabling users to track and analyze the data movement between GPUs. It offers different levels of granularity to allow for both a coarse-grained overview of the system and a detailed view of a specific object or device.

- **System view.** Figure 2A shows the system-wide overview of communication performed by each device and between each pair of devices. The data is displayed both as an interactive graph ① and as a heatmap ③, and can be exported in textual format. Each graph node represents a device, while the edges between the nodes reflect the performed data movement. The information can be shown in terms of both the number of data transfers as well as the volume of transferred data in bytes ②.
- **Object view.** Figure 2B presents a representation of data objects on multiple granularities so that the number of operations for each specific memory address can be tracked. As can be seen in ①, each GPU has its separate object view tab. On each tab, there are objects' address spaces represented as a 1D heatmap, where the information about the memory operations on each address can be viewed by hovering over the heatmap cells ②. This information includes the memory address, types of instructions performed, the remote GPU IDs participating in the communication, and source code line numbers of the memory operations. Additionally, for each object, a 2D view ③ option is available. In 2D view, the user can provide the desired dimensions.
- **Code view.** Figure 2C shows a view that allows to pinpoint the source code lines of the captured inter-GPU remote memory operations. It provides information about the amount of communication performed at each line ①. Code lines where communication is being performed are highlighted and the total amount of communication is shown as a percentage of the total number of data transfers. Upon clicking on a line that is involved in communication, a detailed view shown on the sidebar ② reveals the total number of remote memory operations for each line as well as a system-wide communication heatmap ③ similar to the one provided in System View. Additionally, since multiple objects can be accessed in a single line, information about the objects involved in communication is provided on the sidebar as well.
- **Device view.** Figure 2D represents a brief overview of the remote memory operation types (load/store) and the accessed

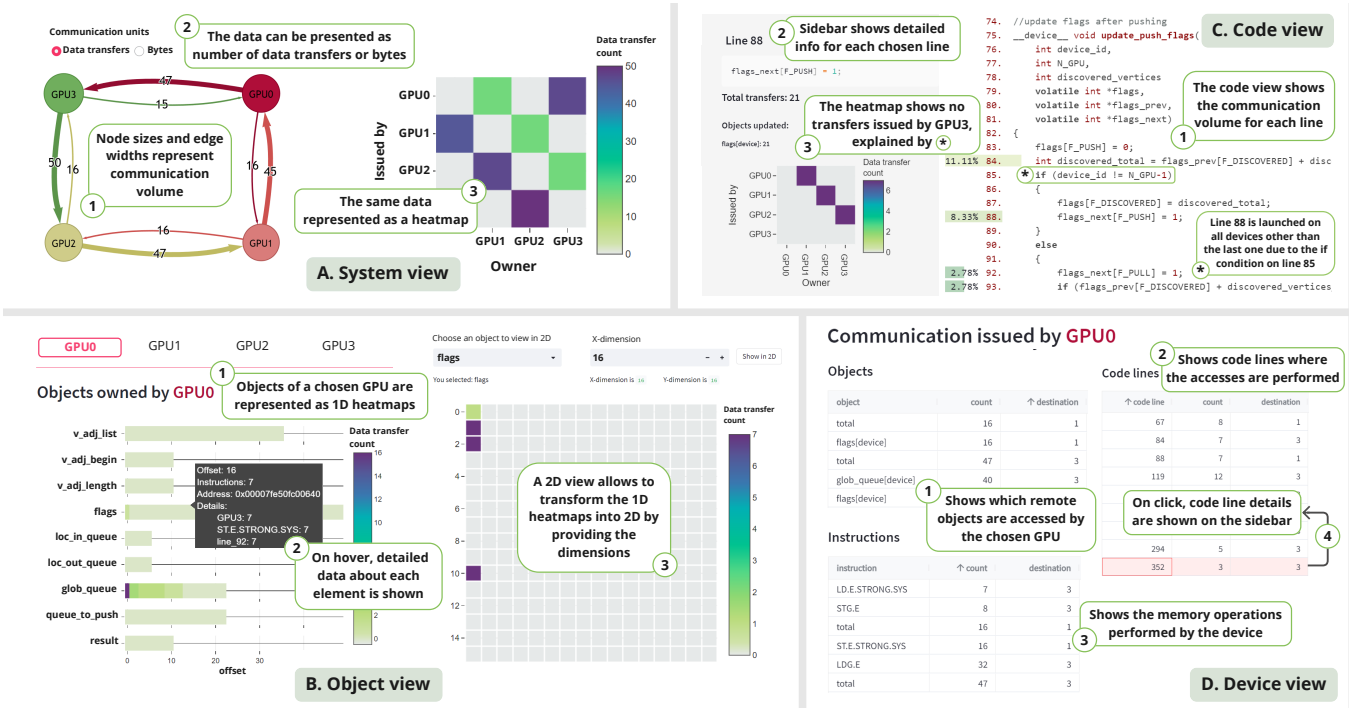


Figure 2: An example of Snoopie's multi-view visualizer. Figures are available under CC-BY [28].

remote objects for each device. While the object view displays objects allocated on a selected GPU and the details regarding remote accesses of these objects by other devices, the device view complements it by instead showing the data transfers performed by the selected GPU itself. It contains object ① and instruction ② tables, representing the data access counts of remote objects and the performed instruction types. Moreover, it includes the source code line numbers ③ to facilitate debugging. On click, a code line number opens the detailed view of the code line ④, the same as the one in the Code View.

4 IMPLEMENTATION OF SNOOPIE

Figure 3 shows the workflow of Snoopie. It starts by instrumenting the executable, such that subsequent CUDA runtime calls will go through Snoopie first, which is then fed to the CUDA runtime and its driver. When a kernel executes on the GPU, Snoopie's instrumented code inspects the remote load and store instructions, which are then registered to a message buffer in the global memory of the device. Once the buffer is full, it is flushed to the host for processing. Snoopie attributes remote accesses to the source/destination device and associates the effective addresses with the data object in the executable. For other multi-GPU communication methods beyond device-initiated direct access, Snoopie provides support for host-initiated transfers. To capture these transfers, Snoopie instruments the `cudaMemcpy*` family of functions. As these functions are executed on the host, no further device instrumentation is necessary. In the final step, Snoopie generates a log file containing trace information for offline analysis and visualization.

To develop Snoopie, one can utilize profiling interfaces like CUPTI API[46], Computer Sanitizer[40], or NVBit [67] designed for NVIDIA GPUs for data movement analysis. Key requirements for Snoopie include instrumenting direct memory accesses and tracing CUDA runtime events. CUPTI can trace CUDA runtime events through its lightweight sampling API, but it lacks in capturing direct memory accesses. On the other hand, Compute Sanitizer, based on binary instrumentation, can monitor load/stores, including remote ones, but it operates at a module level and not at a function or kernel level. NVBit, a dynamic binary instrumentation framework, excels in granularity, allowing precise instrumentation of specific kernels or functions instead of entire modules. Its ability to trace CUDA API events and kernel launches aligns well with our requirements, making it a more suitable choice for Snoopie.

4.1 Capturing P2P Communication

To profile device-initiated remote memory operations, Snoopie relies on NVBit [67], enabling the insertion of instrumentation code into application binary code. NVBit provides built-in functions that disassemble binary instructions, extracting details about the executed operations. Utilizing this information, we identify memory accesses, discerning between reads and writes, and capturing effective addresses. Illustrated in Figure 3 (① and ②), for each identified memory access instruction, we inject a function to record the extracted information in a buffer. This injected function is invoked just before load/store operations, facilitating inspection (④). Subsequently, local addresses are disregarded through address filtering (⑤), allowing only remote addresses to proceed. In the next step,

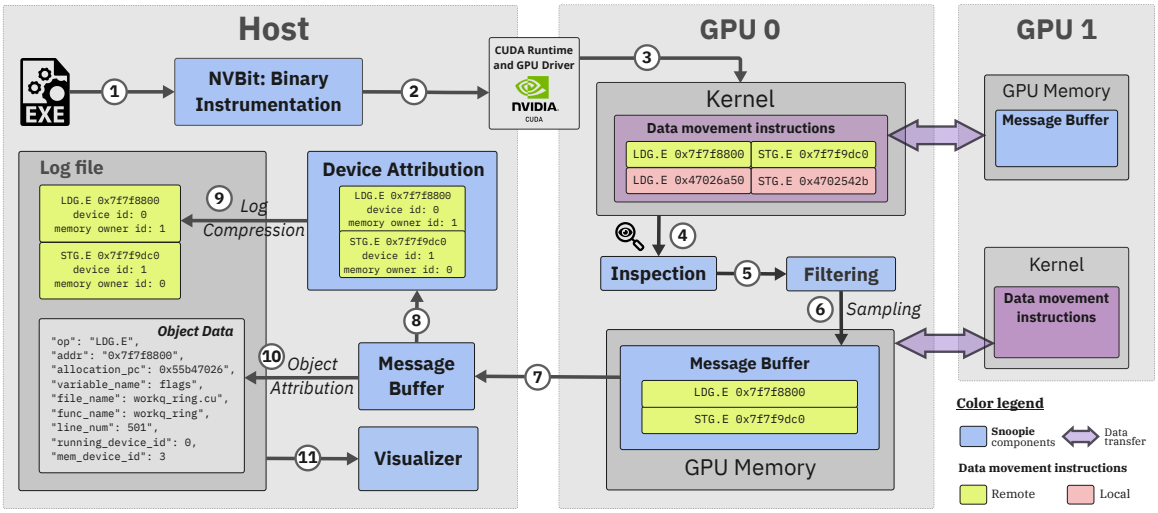


Figure 3: Overview of Snoopie and its components. Figures are available under CC-BY [28].

filtered addresses are sampled if the sampling feature is enabled (⑥).

A channel mechanism in NVBit facilitates the communication of captured information to the host. To minimize data movement overhead, we establish the channel with double buffers in device memory. Not all threads contribute to the channel; instead, within each warp, a single thread compiles memory addresses targeted by memory operations from all other threads in the warp, creating a single message for the entire warp. Thus, each message encapsulates up to 32 memory locations, corresponding to the warp size. Upon channel saturation, the buffer is switched, and the full buffer is transmitted to host memory using a flag implementation that triggers a `cudaMemcpyAsync` operation, depicted in ⑦.

4.2 Device Attribution

The message buffer contains a list of remote addresses accessed during kernel execution, which need to be mapped to corresponding device IDs. The device attribution in Snoopie determines the device to which a pointer value belongs. This capability is facilitated by the support for Unified Virtual Addressing (UVA) provided by NVIDIA. UVA enables device pointers for different devices on the same node to share a common address space. Consequently, the `cuMemAlloc` operations utilize the same address space for pointer values allocated across various devices. This shared address space ensures that if a pointer region is associated with device *A*, any subsequent allocation on device *B* will not return an address overlapping with the region allocated on device *A*. This inherent property allows for the clear identification of the device to which a pointer belongs. Snoopie captures memory ranges allocated for each device whenever a `cuMemAlloc` call is issued by the executable to the CUDA driver and the device ID of the context in which the call is made.

To perform device attribution, Snoopie parses each message received via the message buffer, as can be seen in Figure 3 in ⑧, and determines the region to which each pointer belongs by comparing it with the data obtained from the instrumented `cuMemAlloc` operations. Load/store operations that are a null or empty pointer

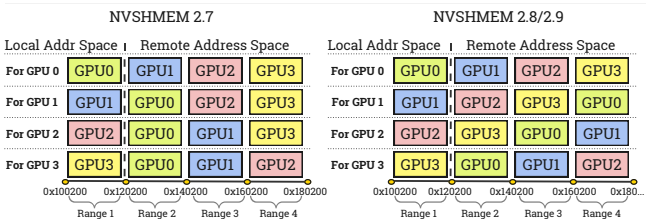


Figure 4: NVSHMEM address space. Figures are available under CC-BY [28].

value are ignored. This is because some threads within a given warp might not be performing a memory operation. The filtered information is then converted into CSV format and is fed into a Zstandard compression stream before it is pushed to the log to be used by the visualizer.

Snoopie uses the approach described above instead of relying on `cuPointerGetAttribute` to obtain information about pointers received from the device. While this CUDA API can provide useful information about a pointer, including its device, base address, and memory type, it may not be able to determine the device ordinal if the memory pointed to by the pointer has already been freed. Our alternative approach enables Snoopie to overcome this limitation by leveraging memory range to device mappings, which persist beyond the lifetime of the memory allocation itself. However, for host-initiated transfers, Snoopie employs `cuPointerGetAttribute` for device attribution because Snoopie can guarantee that the instrumentation of `cudaMemcpy*` functions occurs before any pointer is deallocated, allowing for accurate device attribution in these scenarios.

4.3 NCCL and NVSHMEM Support

As NCCL and NVSHMEM are becoming more popular for scaling deep learning and HPC applications to multiple GPUs, profiling

their communication is essential. NCCL performs communication using device-initiated direct accesses, as a result, Snoopie can support NCCL directly without requiring special handling. Even though NVSHMEM also utilizes device-initiated direct accesses for many of its communication calls, it demands special handling due to its symmetric memory layout and the way peers communicate with each other.

In NVSHMEM, each device has a base address for its address space, and it designates a portion of this address space for each peer. However, the designations for each peer's address space may not match between peers. Figure 4 illustrates the virtual address spaces of each GPU to access other peers within the same node for NVSHMEM v2.7, 2.8 and 2.9. For example, in NVSHMEM 2.7, accessing GPU2 from GPU1 would appear as a load/store operation in Range 3, while accessing the same variable on GPU2 from GPU3 would appear as a load/store operation in Range 4. This method of segmenting the address space makes it non-trivial for our proposed approach to work, as multiple addresses point to the same physical address.

To resolve this issue and add device attribution support for NVSHMEM, Snoopie captures the identifier of the NVSHMEM Processing Element (PE) responsible for executing the direct access operation. An NVSHMEM PE refers to the process that forms an NVSHMEM job. By utilizing the PE identifier along with the captured memory address, Snoopie is able to determine the device on which the respective memory address resides by taking the address spacing rules illustrated in Figure 4 into account.

4.4 Source Code Line Attribution

To associate inter-GPU memory operations to their respective locations in kernel source code, Snoopie employs NVbit's `nvbit_get_line_info` function to extract line information from each instrumented binary instruction. However, it should be noted that NVBit faces limitations in extracting this information from a significant portion of instructions. To overcome this limitation, Snoopie additionally utilizes `nvdisasm` [45] to obtain mapping information between PTX assembly instructions and CUDA source code lines. Given that NVBit supports the disassembly of instrumented instruction binaries into their SASS assembly instructions, Snoopie leverages this mapping information by matching a disassembled SASS instruction with its PTX version, which has been previously mapped to a source code line in the output of `nvdisasm`. This pattern matching between SASS and PTX instructions is only performed when `nvbit_get_line_info` fails to extract the code line information.

4.5 Data Object Attribution

To identify data objects that are accessed during communication, we developed Snoopie with the data object attribution ability. This feature consists of two phases: (1) object recording phase, and (2) object mapping phase.

4.5.1 Object recording phase. The object recording phase is illustrated in Figure 5. During the application execution, Snoopie intercepts function calls for dynamic memory allocations such as `cudaMalloc` and `cudaMallocHost` using a callback function that is inserted to the address space of the profiled process using the

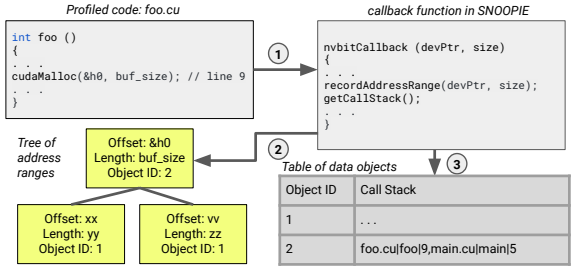


Figure 5: Object recording phase of data object attribution. Figures are available under CC-BY [28].

`LD_PRELOAD` utility[39] in Linux. This callback function calls the original allocation function and records the offset address and byte length of the memory region in a global splay tree (*tree of address ranges*), where each node corresponds to an address range generated by the memory allocation call as shown in Figure 5 ②. In addition to these pair of offset and byte lengths, each node also records an object ID that uniquely identifies the call stack information of the memory allocation call.

We define a data object in our profiling as the call stack of a memory allocation call, encompassing information about the file name, function name, and line number. According to our definition, a data object can be associated with multiple non-contiguous memory regions. This situation arises when there are multiple memory allocation calls sharing the same call stacks, such as a `cudaMalloc` being repetitively called within a loop. To mitigate redundancy in the *tree of address ranges*, we store the details of memory allocation call stacks in a separate hash table, the *table of data objects*, where each entry corresponds to a unique call stack, identified by an object ID that serves as the primary key.

To enhance the debugging process, we have introduced custom wrapper allocation functions enabling programmers to include the variable name, associated with the address range, as a function argument. With this additional information, Snoopie incorporates variable names into the visualization, aiding users in a clearer understanding of object attribution.

4.5.2 Object mapping phase. Snoopie intercepts memory operations during the runtime of profiled applications, logging the memory addresses accessed in these operations. Upon the completion of an application run, Snoopie processes the logged data. At this stage, each recorded memory address undergoes a query in the tree of address ranges to determine the corresponding address range it belongs to. Once Snoopie identifies the relevant node in the tree, it utilizes the object ID within the node to query the corresponding object in the *table of data objects*. Subsequently, Snoopie includes the retrieved entry in the log file.

4.6 Support for Profiling Python Programs

Snoopie supports communication detection for Python applications, with specific considerations for code line and object attributions. These attributions are presently available for Python codes

using Numba [9] or PyTorch [9]. For code line attribution in Numba-decorated functions, Snoopie leverages Numba's feature that generates line info in dispatched CUDA kernels. By enabling this feature, Snoopie can extract line info for each instruction using NvBit's `nvbit_get _line_info`. To extend support for source code line attribution in PyTorch programs, Snoopie captures the call stacks of profiled Python code during the interception of NCCL kernel launches within PyTorch. This information enables Snoopie to attribute the communication detected in the launched NCCL kernels to the specific lines in the Python code that trigger the kernel launches.

For data object attribution, Snoopie logs the allocated address ranges by instrumenting calls to the memory allocation functions of Numba and PyTorch. To enable the instrumentation of Python function calls within Snoopie, we employ the pybind11 library and the C API of Python. This combination facilitates interoperability between Python libraries and the C++ code of Snoopie.

4.7 Reducing Profiling Overhead

Binary instrumentation and monitoring every memory access not surprisingly leads to runtime overhead. We implement several methods to lower this overhead.

- **On-Device Filtering**: Initially, Snoopie transmitted all memory operation records from the instrumented functions to the host. Subsequently, the host filtered these records to retain only the remote accesses. However, this approach posed challenges for certain applications, like stencil, where non-remote memory operations significantly outnumbered remote ones. To overcome this issue, Snoopie performs the filtering directly on the device-side and sends only the remote records to the host, eliminating unnecessary data movement. The filtering is done by the first active thread within the warp, responsible for placing addresses into the communication channel with the host. Instead of placing all addresses onto the channel, the warp's first active thread checks each address against an on-device maintained list of memory allocations that occurred on the host and the devices they belong to. If a remote address is found, the group of addresses are placed on the channel. Otherwise, the addresses are discarded.
- **Sampling**: To minimize runtime overhead and decrease log file size further, we incorporated an optional sampling technique for capturing remote memory operations. This method is integrated into the callback function, which is triggered when a remote memory operation takes place on a GPU. Within the callback function, a pseudo-random number is generated and compared to a threshold specified by the user. If the generated number is lower than the threshold, the callback function records the intercepted memory operation and sends it to the message buffer, as shown in ⑥ in Figure 3. Conversely, if the generated number exceeds the threshold, the intercepted operation is discarded. Note that sampling is performed after on-device filtering so as not to adversely affect the accuracy.
- **Log Compression**: As the tool can monitor and record millions of memory operations, we utilize compression to reduce the size of the logs. This results in a significant reduction, shrinking the accumulating logs from 0.5 gigabytes for 5 million records down to 7 megabytes.

Moreover, the programmer can reduce the overhead by narrowing down the scope of profiling either by selecting a specific device function to monitor or by isolating the communication code to its own device function. A user can also wrap `cudaMalloc` calls for data objects of interest so that Snoopie only monitors those memory regions.

4.8 Discussions on Limitations

Snoopie currently focuses on intra-node multi-GPU setups, meaning it does not support NVSHMEM usage in a multi-node GPU cluster at this time. A similar constraint applies to NCCL applications that utilize MPI; Snoopie only supports NCCL when used within a single OS process. This limitation also impacts future compatibility with PyTorch distributed training applications that rely on NCCL with MPI, underscoring the importance of supporting MPI-utilizing applications in our upcoming efforts. However, this limitation does not apply to NVSHMEM for multi-process single-node usage. Nonetheless, we anticipate no technical obstacles in extending Snoopie to multi-node configurations in the future.

Timeline information falls outside the scope of Snoopie. Its intended use complements the existing GPU tooling landscape, which already includes timeline information through tools like Nsight Systems.

While Snoopie is developed for the CUDA environment at this time, it can be extended to support AMD GPUs. ROCm provides analogous methods of GPU-initiated communication with both P2P RDMA [2] and GPU-centric APIs - RCCL (NCCL equivalent) [4] and ROC_SHMEM [5] (NVSHMEM equivalent), and supports runtime and binary instrumentation [3].

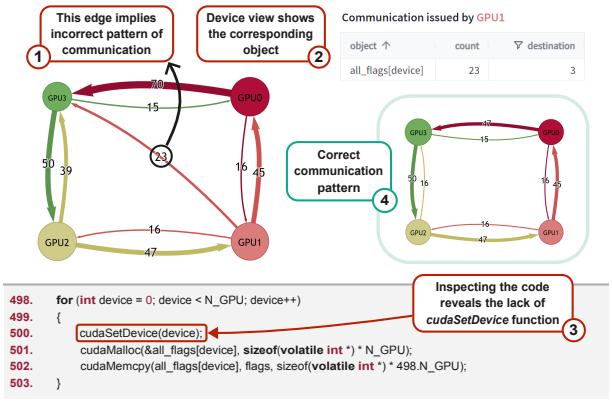
5 EVALUATION

This section evaluates the effectiveness of Snoopie and demonstrates its usability, accuracy, debugging aid, and overhead with four use-cases. The evaluation will focus on Breadth-First Search (BFS), 2D Stencil Jacobi, an AllReduce benchmark from NCCL tests[43], and CosmoFlow [36] The experiments are conducted on a node with 8x Nvidia Ampere 100 GPUs, with an AMD 7763 64-core processor. All the experiments are done without sampling enabled and with device-side filtering unless otherwise stated.

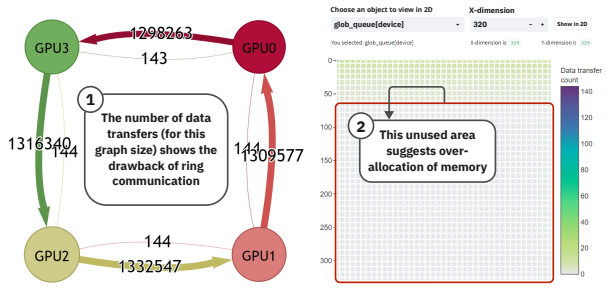
5.1 Breath-First Search (BFS)

BFS traversal is an algorithm that is commonly used to traverse and search graph-like data structures. Despite following a simple principle of traversing nodes level by level, it has irregular communication patterns. Therefore, BFS is particularly useful for analyzing communication. The BFS implementation in use is based on a ring principle and leverages the device-initiated P2P direct remote memory access to communicate between the GPUs. Devices exchange data in a ring manner by reading from globally accessible buffers, while the order is ensured by passing flags.

Demo 1. Observing the communication patterns and debugging. This demonstration is run on a small graph containing only 12 nodes for better traceability. The communication pattern in a system-wide view on Figure 6a shows that mostly the neighboring



(a) Demo 1: Fixing a bug with the help of Snoopie



(b) Demo 2: Analysis of a ring execution model

Figure 6: Use cases of Snoopie: a BFS example. Figures are available under CC-BY [28].

devices are communicating with each other. Yet, there is an arrow ① representing communication between devices 1 and 3, which are not neighbors, thus not expected in a ring communication. The object view and the device view can give further hints on the possible reason for such behavior ②, highlighting an object associated with the excess communication and pointing out that the data is being *read* from GPU3. Closer inspection ③ of the object's allocation reveals that although all the objects are allocated on their corresponding devices correctly, the pointer array to all devices' flags is allocated only on GPU3. Therefore, all the flag updates incorrectly require getting the flag pointer from GPU3 by all the devices in the system. Proper allocation of the pointer array solves the issue ④ and removes the excess transfers.

Demo 2. Performance bottleneck detection. For a use case that involves analysis of issues on a larger scale we use a bigger graph *web-BerkStan* with around 670K nodes. Figure 6b shows the Snoopie output. The resulting amount of communication in the system turns out to be almost twice the number of vertices. The sheer amount of transferred data is not problematic, since communicating the node data requires transferring 2 values: the node index and its depth. However, all communications need to pass through a ring ① yielding excess transfers. This fact suggests usage of direct transfers between the devices with buffers for each device pair at the cost of finding the nodes' owners. Moreover, inspection of the ring buffer in ② shows that the allocated buffer space for this graph

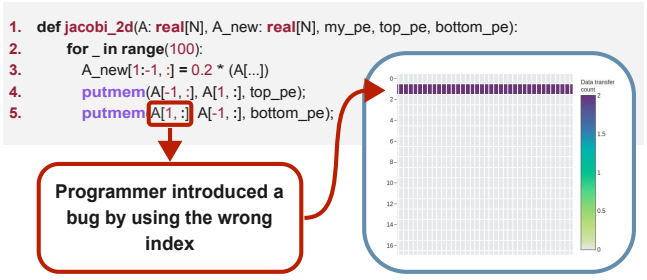


Figure 7: Demo 3: Debugging an erroneous halo region of a 2D stencil computation using Snoopie's object view. Instead of C++, simplified Python code is shown for clarity. Figures are available under CC-BY [28].

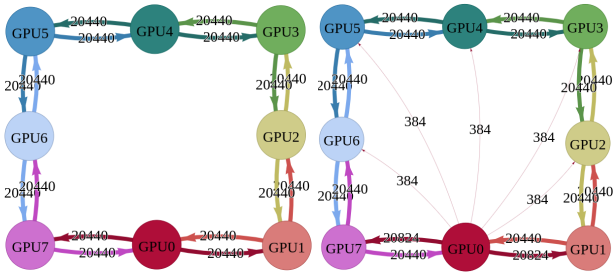
is much larger than required, suggesting memory consumption for communication buffers could be reduced.

5.2 2D Stencil Computation

The stencil computation involves updating the value of each element on a grid based on the values of its neighboring elements, typically within a fixed-size window. These computations are commonly found in scientific applications and image processing. Our evaluation uses a 5-point 2D star stencil operation over a domain of 1024^2 which is divided over 8 GPUs on the y-axis such that each GPU operates on an array of size $1024 * 128$. We experimented with two variants of the kernel; i) *Stencil-p2p* is based on the implementation by NVIDIA [42] that uses the device-initiated P2P communication model across GPUs. ii) *Stencil-NVSHMEM* is same as Stencil-p2p, however, uses NVSHMEM implementation provided by NVIDIA [41].

Demo 3. Using object view to debug communication. To simulate a typical bug in stencil computations, we deliberately introduce an indexing error in the Stencil-p2p code. This error involves changing the halo region in each GPU, which corresponds to the region accessed by the neighboring GPU. The object view of Snoopie demonstrates the data transfers visually and shows all the data objects allocated in each GPU. Notably, there are two one-dimensional arrays, named `A` and `A_new`, which are accessed as 2D arrays in the stencil computation. The intended halo regions for these arrays in each GPU are the first and last rows. However, as depicted in Figure 7, the color-coded remotely accessed region should be the first row of the `A_new` array, but mistakenly, it is indicating the second row.

Demo 4. Comparing communication induced by different stencil implementations. Figures 8a and 8b compare the communication patterns and volumes of the Stencil-p2p and Stencil-NVSHMEM codes on 8 GPUs. Both implementations display nearly the same communication patterns and volumes. The only difference is the small-sized directed edges from GPU 0 to the rest of the GPUs in the NVSHMEM version. This small-volume data traffic is caused by the host-initiated inter-GPU communications using device-to-device `cuMemcpy` function calls.



(a) Stencil-p2p (b) Stencil-nvshmem

Figure 8: Demo 4: Comparing communication pattern and volume across two multi-GPU stencil implementations. Figures are available under CC-BY [28].

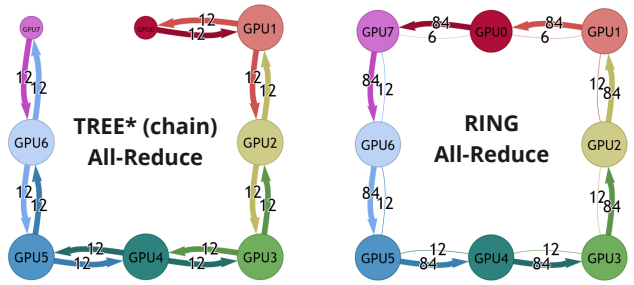
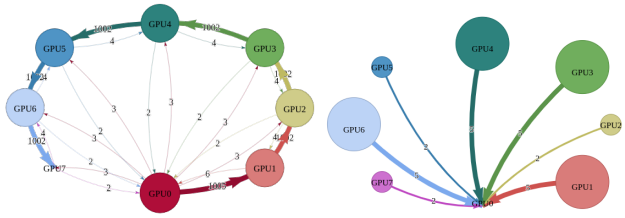


Figure 9: Demo 5: Comparing NCCL All-Reduce with Tree or Ring algorithms. Figures are available under CC-BY [28].

5.3 AllReduce from NCCL

NCCL is a communication library that uses different underlying algorithms depending on the supplied load. In this demo, we focus on one collective communication primitive, namely `AllReduce`, which offers different algorithms including `RING` and `TREE`. The `TREE` algorithm, introduced in NCCL 2.4, is designed to improve scalability and performance for small and medium-sized operations by using a double binary tree which appears as a chain/list when running in a single node to implement `all_reduce`, with each rank sending and receiving $2N$, except for the edges which exchange N where N represents the size of the data being reduced. The `RING` algorithm, on the other hand, transmits data in chunks around the ring, with each rank sending data to the next and receiving data from the previous rank to achieve maximum bandwidth. As NCCL can switch between algorithms based on latency and bandwidth requirements of the target platform, it might be valuable for the user to know which algorithm is being used for the communication operation.

Demo 5. NCCL communication visualization. We ran the `all_reduce` NCCL test code with the output shown in Figure 9. The system view of Snoopie enables users to observe the chain structure when the `TREE` algorithm is used. Note that this is the expected behavior of `TREE` within a single node. The visualizer also displays the communication size of each GPU node, indicating how GPU0 and GPU7 are smaller than the other GPUs when using the `TREE` algorithm, as expected for the ends of the chain. In contrast, the visualizer shows almost equivalent sizes for all nodes when using the `RING` algorithm.



(a) Data Parallel (b) Model Parallel

Figure 10: Demo 6: PyTorch implementations of Cosmoflow utilizing different parallelization strategies. Figures are available under CC-BY [28].

5.4 CosmoFlow

CosmoFlow is a distributed deep learning application built to predict cosmological parameters given 3D matter distribution data [36]. In order to ensure compatibility with Snoopie and to show different possible communication patterns, we port the PyTorch model to use a single process and implement two versions - Data Parallel and Model Parallel.

Demo 6: Comparing data vs model parallelism. The Data Parallel implementation utilizes built-in automatic parallelism functionality in PyTorch and splits the model and the data across devices. The communication generated uses NCCL and we observe a balanced chain shown in Figure 10a.

Our naive Model Parallel implementation, on the other hand, requires manual assignment of model layers to devices and is more susceptible to issues with balancing. Our port uses a naive strategy of sequentially assigning layers to subsequent devices. In contrast to the Data Parallel implementation that splits data across devices, the Model Parallel implementation requires the placement of all tensors in a single GPU, which in our case is GPU 0. The centralized location of allocated memory in a single GPU causes the model layers in all of the other devices to read/write into the memory of GPU 0 as shown by the pattern in Figure 10b. Through this experiment, we note that Snoopie can greatly aid in mapping communication among devices and comparing the communication patterns between different parallelization strategies.

5.5 Overhead and Accuracy

Snoopie is accurate, but being an instrumentation-based tool, it introduces runtime overhead to the profiled application. Figure 11 shows the performance slowdown compared to the baseline without instrumentation. Note that Overhead measurements include the log compression time. The overhead of Snoopie is much higher on NCCL due to the huge number of memory allocations on GPUs that NCCL does in its implementation of collective communication functions. Since Snoopie captures the allocated address ranges to detect inter-GPU remote accesses, this high number of memory allocations affects the profiling overhead. Performing device-side filtering of the remote operation results in a 1.8× improvement

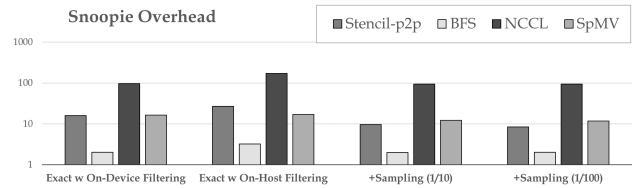


Figure 11: Overhead comparison of Snoopie compared to uninstrumented code. The Y-axis is log-scaled. Figures are available under CC-BY [28].

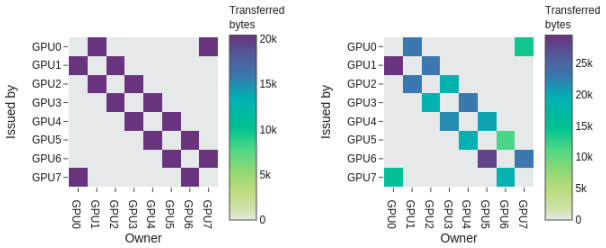


Figure 12: a) Exact profiling, b) Sampling-based profiling. Figures are available under CC-BY [28].

compared to filtering on the host, without compromising the profiling accuracy. This optimization also applies to other use cases, albeit to a lesser extent. Specifically, when the device-side filtering optimization is applied, the overhead of Snoopie is 16×, 2.04×, 96×, and 16.4× for Stencil, BFS, and NCCL, respectively.

To further reduce overhead, sampling-based profiling can be employed but may lower Snoopie's accuracy. This approach shows much more impact on stencil than the other benchmarks due to Snoopie's memory-intensive nature and numerous local loads/stores, which Snoopie discards when these accesses are not sampled. While the overhead of on-device filtering, 1/10 sampling rate, and 1/100 sampling rate are nearly the same for BFS, NCCL, and SpMV, the overhead on stencil shows gradual reduction with an overhead of 9.6× on 1/10 sampling rate and 8.5× on 1/100 sampling rate.

There are cases in which the majority of the overhead was caused by the massive size of communication reported and the processing and logging following it. In these cases, adjusting the sampling parameter and the profiling method did not achieve notable improvements, as the overhead ranged between 10× and 20× though we argue that the real execution time is reasonable.

Next, we evaluate the accuracy of Snoopie in capturing communication when monitoring the Stencil-p2p code with and without sampling. Figure 12 displays the captured communication matrix with the exact profiling and when the sample size is 10% of all memory operations. The patterns are similar though the variation of color indicates that the communication volumes among GPU pairs are less balanced in the sampling-based profiling, which results from the stochastic nature of the sampling mechanism.

The forthcoming efforts will concentrate on improving the overhead of Snoopie while preserving its accuracy. Additionally, we offer both exact and sampling profiling modes, enabling programmers to choose between more detailed examination of the code region using exact profiling or less accurate sampling-based profiling.

6 RELATED WORK

Table 1 summarizes the comparison between Snoopie and the other communication profilers that work on GPUs. Nsight Systems [48] and NVTAGS [49] allow programmers to profile multi-GPU communications. Tools like Eztrace [64], Extrae[12], and Score-P[31] generate traces of host-device and device-device data movements, however, all of these tools are limited to capture only explicit memory and the latter three tracing tools support neither NCCL nor NVSHMEM. ComDetective captures inter-thread communication but only for multicore CPUs [60]. ComScribe introduced in [8] and extended in [63], captures P2P host-device, P2P device-device, and collective communications among multiple GPUs. It only presents the profiling results in the form of communication matrices.

Diogenes [68] leverages the feed-forward measurement profiling model to identify problematic synchronization and data transfers through multi-stage/multi-run profiling of GPU applications. However, this tool does not capture inter-GPU remote memory operations. Li et al. [34] developed Tartan, which is a benchmark suite that characterizes scale-up and scale-out multi-GPU applications. Using this benchmark suite, they investigated the latency and bandwidth of various GPU bottlenecks. Another work by Pearson et al. [56] introduced Comm|Scope, which is another benchmark suite that characterizes communications in multi-GPU machines.

Paraver [15] is a visualization tool for parallel programs running PVM message passing library [20] that translates the trace files analyzed by DIMEMAS simulator [21] into graphical displays. VAMPIR [38] is a graphical tool for MPI applications. It converts an input trace file into various graphical system views, which include the current activity of each processor, a timeline system view, and summary statistics of system behavior. PerfExplorer [25] presents analysis results on parallel applications. It performs data mining on raw performance data stored in a PerfDMF DBMS [26] and presents the results in the forms of performance graphs, correlation scatterplots, and summary statistics. EXTRAVIS[16] is a tool that visualizes execution traces in order to aid its users in comprehending the profiled applications.

HPCToolkit[1] presents its profiling results in a code-centric way using its graphical user interface, HPCViewer. It visualizes the source code of the profiled application and displays the number of events sampled using PMUs from each source code line. PerformanceHat [14] is another tool that presents the source code view of profiled applications in an IDE integration. It augments the view with monitoring data generated during the production run of the application and creates a performance model. Kousha et al. [32] devised a visualization tool that leverages CUPTI [47] and MPI_T [62] to capture the utilization of GPU interconnects and correlates it with MPI communication patterns. Schaad et al.[61] developed a performance visualization tool that reports data movement and reuse behavior of profiled applications by leveraging static dataflow analysis. In contrast, our visualizer stands apart by offering fine-grained visualization of GPU communication.

GVPROF [69] and ValueExpert [70] use binary instrumentation for analyzing value-related inefficiencies in GPU kernels. DrGPUM [35] employs binary instrumentation to detect inefficiencies related to memory usage. While these tools attribute identified inefficiencies to data objects and code lines, it's important to note that they are designed for single-GPU applications, unlike Snoopie.

7 CONCLUSION

Our work introduces Snoopie, a profiling tool that monitors and analyzes multi-GPU communication within a single node. Snoopie stands out from previous tools by being able to capture communication triggered by P2P direct accesses and NCCL/NVSHMEM library calls. To help locate performance bottlenecks and facilitate debugging, Snoopie supports communication attribution to the involved devices, source code lines, and data objects. The tool presents this information in a visualizer with support for multiple levels of granularity. We demonstrated its capabilities through various use cases. Future work will focus on increasing the tool's user-friendliness, adding support for multi-node communication, and applying it to a wider range of applications.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under Grant 949587 and in part by the Royal Society-Newton Advanced Fellowship under Grant NAF\R2\202207.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* 22, 6 (apr 2010), 685–701.

[2] Inc Advanced Micro Devices. 2024. AMD ROCm Documentation. https://rocm.docs.amd.com/.

[3] Inc Advanced Micro Devices. 2024. Omnitrace. https://github.com/AMDResearch/omnitrace.

[4] Inc Advanced Micro Devices. 2024. ROCm RCCL Documentation. https://rocm.docs.amd.com/projects/rccl/en/latest/.

[5] Inc Advanced Micro Devices. 2024. ROCSHMEM. https://github.com/ROCm-DeveloperTools/ROC_SHMEM.

[6] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2017. Offloading Communication Control Logic in GPU Accelerated Applications. In *Proceedings of the 17th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing (CCGrid'17)*. IEEE, New York, NY, USA, 248–257. https://doi.org/10.1109/CCGRID.2017.29

[7] Meta AI. 2024. PyTorch. https://github.com/pytorch/pytorch.

[8] Palwisha Akhtar, Erhan Tezcan, Fareed Mohammad Qararyah, and Didem Unat. 2020. ComScribe: Identifying Intra-Node GPU Communication. In *Benchmarking, Measuring, and Optimizing: Third BenchCouncil International Symposium, Bench'20*. Springer-Verlag, Berlin, Heidelberg, 157–174. https://doi.org/10.1007/978-3-030-71058-3_10

[9] Continuum Analytics. 2024. A Just-In-Time Compiler for Numerical Functions in Python. https://github.com/numba/numba.

[10] Google Brain. 2024. TensorFlow. https://github.com/tensorflow/tensorflow.

[11] U.V. Catalyurek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 673–693. https://doi.org/10.1109/71.780863

[12] Barcelona Supercomputing Center. [n. d.]. BSC-Performance-Tools: Extrae. https://tools.bsc.es/extrae.

[13] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. 2022. Scalable Irregular Parallelism with GPUs: Getting CPUs out of the Way. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE, New York, NY, USA, Article 50, 16 pages.

[14] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive Production Performance Feedback in the IDE. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 971–981. https://doi.org/10.1109/ICSE.2019.00102

[15] Departament Computadors, Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. PARAVER: A tool to visualize and analyze parallel code. *WoTUG-18* 44 (03 1995).

[16] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. 2007. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 49–58. https://doi.org/10.1109/ICPC.2007.39

[17] Jack J. Dongarra. 2022. The Evolution of Mathematical Software. *Commun. ACM* 65, 12 (Nov 2022), 66–72. https://doi.org/10.1145/3554977

[18] Apache Software Foundation. 2024. Apache MXNet for Deep Learning. https://github.com/apache/mxnet.

[19] Python Software Foundation. 2024. Python/C API Reference Manual. https://docs.python.org/3/c-api/index.html.

[20] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. 1995. Pvm 3 User's Guide And Reference Manual. (11 1995).

[21] Sergi Girona, Toni Cortes, and Vincent Pillet. 1994. Effect of Short Term Scheduling on Message Passing Multiprogrammed Systems. (11 1994).

[22] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. Institute for Electrical and Electronics Engineers, New York, NY, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

[23] Khaled Hamidouche and Michael LeBeane. 2020. GPU INitiated OPenSHMEM: Correct and Efficient Intra-Kernel Networking for DGPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. ACM, New York, NY, USA, 336–347. https://doi.org/10.1145/3332466.3374544

[24] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. 2015. Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters. In *2015 IEEE International Conference on Cluster Computing*. IEEE, New York, NY, USA, 78–87. https://doi.org/10.1109/CLUSTER.2015.21

[25] K.A. Huck and A.D. Malony. 2005. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 41–41. https://doi.org/10.1109/SC.2005.55

[26] K.A. Huck, A.D. Malony, R. Bell, and A. Morris. 2005. Design and implementation of a parallel performance data management framework. In *2005 International Conference on Parallel Processing (ICPP'05)*. 473–482. https://doi.org/10.1109/ICPP.2005.29

[27] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs Are Not in Charge. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. ACM, New York, NY, USA, 192–202. https://doi.org/10.1145/3577193.3593713

[28] Mohammad Kefah Taha Issa, Didem Unat, Dogan Sagbili, Muhammad Aditya Sasongko, Ilyas Turimbetov, and Javid Baydamirli. 2024. Snoopie Figures. https://doi.org/10.6084/m9.figshare.c.7190766.v1

[29] Wenzel Jakob. 2024. pybind11 — Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

[30] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[31] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[32] Pouya Kousha, Bharath Ramesh, Kaushik Kandadi Suresh, Ching-Hsiang Chu, Arpan Jain, Nick Sarkauskas, Hari Subramoni, and Dhabaleswar K. Panda. 2019. Designing a Profiling and Visualization Tool for Scalable and In-depth Analysis of High-Performance GPU Clusters. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 93–102. https://doi.org/10.1109/HiPC.2019.00022

[33] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2017. GPU Triggered Networking for Intra-Kernel Communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/3126908.3126950

[34] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[35] Mao Lin, Keren Zhou, and Pengfei Su. 2023. DrGPUM: Guiding Memory Optimization for GPU-Accelerated Applications. In *Proceedings of the 28th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) *(ASPLOS 2023)*. ACM, New York, NY, USA, 164–178. https://doi.org/10.1145/3582016.3582044

[36] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor Lee. 2019. CosmoFlow: using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 65, 11 pages. https://doi.org/10.1109/SC.2018.00068

[37] John McCalpin. 2016. STREAM: Sustainable Memory Bandwidth in High Performance Computers. HPCWire https://www.hpcwire.com/2016/11/07/mccalpin-traces-hpc-system-balance-trends.

[38] Wolfgang Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and analysis of mpi resources. *Supercomputer* 12 (05 1996).

[39] Greg Nakhimovsky. 2001. Debugging and Performance Tuning with Library Interposers. http://dsc.sun.com/solaris/articles/lib_interposers.html.

[40] NVIDIA. 2022. Compute Sanitizer. https://docs.nvidia.com/cuda/compute-sanitizer/

[41] NVIDIA. 2022. Multi GPU Programming Models NVSHMEM. https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/nvshmem_opt.

[42] NVIDIA. 2022. Multi GPU Programming Models P2P. https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/multi_threaded_p2p.

[43] NVIDIA. 2022. NCCL Tests. https://github.com/NVIDIA/nccl-tests.

[44] NVIDIA. 2022. Nvidia OpenSHMEM Library (NVSHMEM) documentation. https://docs.nvidia.com/nvshmem/api/

[45] NVIDIA. 2023. CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[46] NVIDIA. 2023. CUPTI. https://docs.nvidia.com/cupti/

[47] NVIDIA. 2023. NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit. https://developer.nvidia.com/cupti

[48] NVIDIA. 2023. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[49] NVIDIA. 2023. NVTAGS. https://developer.nvidia.com/nvidia-nvtags.

[50] NVIDIA. 2023. NVTX. https://docs.nvidia.com/nvtx/.

[51] NVIDIA. 2023. Parallel Thread Execution ISA Version 8.2. https://docs.nvidia.com/cuda/parallel-thread-execution/

[52] NVIDIA. 2023. Parallel Thread Execution ISA Version 8.2. https://docs.nvidia.com/cuda/parallel-thread-execution/

[53] Nvidia. 2024. NCCL. https://github.com/NVIDIA/nccl.

[54] Lena Oden and Holger Fröning. 2013. GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters. In *2013 IEEE Int'l Conference on Cluster Computing (CLUSTER)*. IEEE, Indianapolis, IN, USA, 1–8. https://doi.org/10.1109/CLUSTER.2013.6702638

[55] Marc S. Orr, Shuai Che, Bradford M. Beckmann, Mark Oskin, Steven K. Reinhardt, and David A. Wood. 2017. Gravel: Fine-Grain GPU-Initiated Network Messages. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado, USA) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/3126908.3126914

[56] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/3297663.3310299

[57] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. 2011. *OpenSHMEM - Toward a Unified RMA Model*. Springer US, Boston, MA, 1379–1391. https://doi.org/10.1007/978-0-387-09766-4_490

[58] Sreeram Potluri, Anshuman Goswami, Davide Rossetti, C.J. Newburn, Manjunath Gorentla Venkata, and Neena Imam. 2017. GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, New York, NY, USA, 253–262. https://doi.org/10.1109/HiPC.2017.00037

[59] Microsoft Research. 2024. deepspeed. https://github.com/microsoft/DeepSpeed.

[60] Muhammad Aditya Sasongko, Milind Chabbi, Palwisha Akhtar, and Didem Unat. 2019. ComDetective: a lightweight communication detection tool for threads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 18, 21 pages. https://doi.org/10.1145/3295500.3356214

[61] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2022. Boosting Performance Optimization with Interactive Data Movement Visualization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 64, 16 pages.

[62] Martin Schulz. [n. d.]. MPIT: A New Interface for Performance Tools in MPI 3. http://cscads.rice.edu/2010-08-cscads-mpit.pdf

[63] Muhammet Abdullah Soytürk, Palwisha Akhtar, Erhan Tezcan, and Didem Unat. 2022. Monitoring Collective Communication Among GPUs. In *Euro-Par 2021: Parallel Processing Workshops*, Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci (Eds.). Springer International Publishing, Cham, 41–52.

[64] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. 2011. EZTrace: A Generic Framework for Performance Analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 618–619. https://doi.org/10.1109/CCGrid.2011.83

[65] James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO) *(SC'23)*. ACM, New York, NY, USA, Article 31, 13 pages. https://doi.org/10.1145/3581784.3607046

[66] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[67] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[68] Benjamin Welton and Barton P. Miller. 2019. Diogenes: Looking for an Honest CPU/GPU Performance Measurement Tool. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 21, 20 pages. https://doi.org/10.1145/3295500.3356213

[69] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2020. GVPROF: A Value Profiler for GPU-Based Clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. https://doi.org/10.1109/SC41405.2020.00093

[70] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: exploring value patterns in GPU-accelerated applications. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. ACM, New York, NY, USA, 171–185. https://doi.org/10.1145/3503222.3507708