ComScribe: Identifying Intra-node GPU Communication

Palwisha Akhtar[0000−0003−0279−031X], Erhan Tezcan[0000−0001−5129−4166], Fareed Mohammad Qararyah[0000−0002−3955−2836], and Didem Unat[0000−0002−2351−0770]

Department of Computer Science and Engineering, Koç University, Turkey
{pakhtar19,etezcan19,fqararyah18,dunat}@ku.edu.tr

Abstract. GPU communication plays a critical role in performance and scalability of multi-GPU accelerated applications. With the ever increasing methods and types of communication, it is often hard for the programmer to know the exact amount and type of communication taking place in an application. Though there are prior works that detect communication in distributed systems for MPI and multi-threaded applications on shared memory systems, to our knowledge, none of these works identify intra-node GPU communication. We propose a tool, ComScribe that identifies and categorizes types of communication among all GPU-GPU and CPU-GPU pairs in a node. Built on top of NVIDIA's profiler *nvprof*, ComScribe visualizes data movement as a communication matrix or bar-chart for explicit communication primitives, Unified Memory operations, and Zero-copy Memory transfers. To validate our tool on 16 GPUs, we present communication patterns of 8 micro- and 3 macro-benchmarks from NVIDIA, Comm|Scope, and MGBench benchmark suites. To demonstrate tool's capabilities in real-life applications, we also present insightful communication matrices of two deep neural network models. All in all, ComScribe can guide the programmer in identifying which groups of GPUs communicate in what volume by using which primitives. This offers avenues to detect performance bottlenecks and more importantly communication bugs in an application.

Keywords: Inter-GPU communication · Multi-GPUs · Profiling

1 Introduction

Graphical Processing Units (GPUs) are increasingly becoming common and vital in compute-intensive applications such as deep learning, big data and scientific computing. With just a single GPU and exceeding working sets, memory becomes a bottleneck, inevitably shifting the trend towards multi-GPU computing. Currently, a variety of single node multi-GPU systems are available such as NVIDIA's TU102, V100-DGX2 and DGX A100 with varying number of GPUs ranging from 2 to 16, connected via NVLink based interconnects [15, 19, 20, 26].

Traditionally, multiple GPUs in a node are interconnected via PCIe such as in DGX1 with PCIe Gen3 [19]. A balanced tree structure is formed by the

PCIe network where one GPU is connected to another through a PCIe switch. In such systems, multi-GPU application performance is constrained by the lower bandwidth of PCIe and the indirect connections between GPUs. With the introduction of GPUDirect Peer-to-Peer that enables direct data access and transfer capability among multiple GPUs and GPU-oriented interconnects such as NVLink, programmers can directly read or write local graphics memory, peer GPU memory or the system memory, all in a common shared address space [13]. The higher bandwidth of such interconnects improve the performance of applications utilizing inter-GPU communication.

To handle the communication between multiple GPUs, CUDA API offers various data transfer options to the programmer under the hood of Unified Virtual Addressing (UVA), Zero-copy Memory and Unified Memory paradigms. Explicit data transfers can be carried out in the traditional way using CUDA's memory copy functions but with Zero-copy Memory or Unified Memory operations, the memory management is handled by the CUDA driver and programmer is free from the burden of tracking data allocations and transfers among host and the GPUs in a node. However, in such a scenario, users are unaware of the underlying communication taking place between the GPUs.

In this pool of varying communication types and methods for single node multi-GPU systems, communication remains as a critical programming component and performance contributor [31]. As a result, identifying communication between GPUs can guide the programmer in many aspects. Firstly there are many cases of communication that result in implicit data transfers, and these unforeseen communications will occur if the programmer is indifferent towards the topology and device capabilities. For example, an application might not have any implicit communication with an all-to-all GPU topology, but might have them with a Hyper-cube Mesh topology such as in DGX-1, where some GPUs do not have direct interconnect to some others, and transfers among them may require explicit routing, as NVLink is not self-routed [16]. Secondly, it is possible for there to be bugs in an application such that the result is not affected, however the bug causes an unexpected communication pattern, which may be hard to infer from debugging alone. Finally, monitoring and identifying communication among multiple GPUs can help reason about scalability issues and performance divergence between different implementations of the same application, and guide the programmer to utilize the GPU interconnects for better performance [8]. For instance, a single GPU application when naively scaled up to multiple GPUs, may follow a master-slave communication pattern, which would mean that it is not effectively using the GPU interconnects. All in all, identifying which groups of GPUs communicate in what volume and their quantitative comparison offer avenues to detect performance bugs and tune software for scalability.

Even though there are communication identification tools for MPI applications on distributed systems (e.g. EZTrace [33]) and for multi-threaded applications on shared memory systems (e.g. ComDetective [29]), to our knowledge, there is no communication monitoring tool designed for single node multi-GPU systems. In this work, we propose ComScribe, a tool that can monitor, iden-

tify, and quantify different types of communication among GPU devices. The tool can generate a communication matrix that shows the amount of data movement between two pairs of GPUs or with host. In addition, it can automatically identify the types of communication i.e. explicit transfers using CUDA primitives or implicit transfers using Zero-copy Memory or Unified Memory operations. ComScribe is built on top of NVIDIA's profiling tool *nvprof* [25] that provides a timeline of all activities taking place on a GPU such as kernel execution, memory copy or memory set, data transmitted or received through NVLink. However, *nvprof* does not readily generate communication matrices and the *nvprof* trace consists of extraneous information that is unnecessary for a user who is concerned with communication among GPUs. Our tool overcomes this limitation and works in two steps: First, it collects intra-node multi-GPU and CPU-GPU memory transfer information during execution with *nvprof*. Then, it performs post-processing to quantify communication among GPUs as well as the host, and identify communication types. Our contributions are summarized below:

- We present ComScribe, a tool for generating communication matrices that show both the number of transfers and amount of data transferred between every GPU-GPU and CPU-GPU pair in a node.
- The tool also identifies different types of communication such as explicit data transfers using CUDA primitives or implicit transfers using Zero-copy Memory or Unified Memory.
- We validate our tool against known communication patterns using 11 benchmarks from Comm|Scope [28], MGBench [7] and NVIDIA on a multi-GPU V100-DGX2 system and demonstrate how ComScribe is used for detecting a communication bug in one of these benchmarks.
- We present communication matrices for 2 well-known deep neural network models and demonstrate how ComScribe can be used for explaining different implementations of data parallelism in deep learning.

ComScribe is available at https://github.com/ParCoreLab/ComScribe.

2 Background

In this section we will discuss various data exchange scenarios with CUDA data transfer primitives that are supported by NVIDIA GPUs. Traditionally, for point-to-point CPU-GPU and GPU-GPU communication, programmers initiate explicit data transfers using `cudaMemcpy`. GPU-GPU transfers involve copying through host memory, which not only requires careful tracking of device pointer allocations but may also result in performance degradation [17]. With the introduction of GPUDirect Peer-to-Peer and GPU-oriented interconnects among multiple GPUs, one GPU can directly transfer or access data at another GPU's memory within a shared memory node [18, 13, 16, 15]. For host-initiated explicit peer-to-peer memory copies, `cudaMemcpyPeer` is used, which also requires passing device IDs as arguments, so that CUDA can infer the direction of transfer.

With CUDA 4.0, NVIDIA introduced a new "unified addressing space" mode called Unified Virtual Addressing (UVA), allowing all CUDA execution – CPU

and GPU – in a single address space. Applications using UVA with peer-access do not require device IDs to be specified to indicate the direction of transfer for an explicit host-initiated copy. Instead, by using `cudaMemcpy` with transfer kind `cudaMemcpyDefault`, CUDA runtime can infer the location of the data from its pointer and carry out the copy. However, if the peer access is disabled, implicit transfers through the host will occur, resulting in communication overhead.

Similar to CUDA explicit bulk transfers, data accesses between host and device are also possible with Zero-copy Memory, which requires mapped pinned memory. Pinned allocations on the host are mapped into the unified virtual address space and device kernels can directly access them [24]. Similarly, implicit peer-to-peer data accesses between GPUs are possible during kernel execution in Zero-copy Memory paradigm.

In CUDA 6.0, NVIDIA introduced a single address space called Unified Memory. With data allocated using Unified Memory, CUDA becomes responsible for migrating memory pages to the memory of the accessing CPU or GPU on demand. Pascal GPU architecture via its Page Migration Engine is the first that supports virtual memory page faulting and migration [14]. Unified Memory offers ease of programming as managed memory is accessible to CPU and GPUs with a single pointer and does not require explicit memory transfers among them.

Multi-GPU applications are developed using any of the above CUDA data transfer primitives with respect to the GPU hardware, CUDA version and application model. We cover all types of communication options available for intra-node communication using CUDA, and present a tool for generating a communication matrix and identifying the types of communication for single node multi-GPU applications.

3 Types of Communication

As discussed in the previous section, various data transfer primitives are available in the CUDA programming model, each requiring different software or hardware support. Therefore, we have divided these communication types into two categories: Peer-to-Peer and Host-Device shown in Figure 1. These categories are further divided into sub-categories based on the data transfer options utilized, e.g. explicit transfers using `cudaMemcpy` and `cudaMemcpyPeer`, or implicit transfers using Zero-copy Memory or Unified Memory operations. In order to make reference to the sub-categories easier, we have assigned them a case number. We should also note that synchrony is not a consideration here, because we are interested in the type and amount of the transfer. For example, whether we use `cudaMemcpy` or `cudaMemcpyAsync` does not matter, as both methods will result in the same amount of data transfer, though at different times. In the following subsections, we explain each category and their sub-categories as cases.

3.1 Peer-to-Peer Communication

Peer-to-Peer communication refers to a data transfer between two GPUs that support peer-to-peer memory access. This type of communication depends on

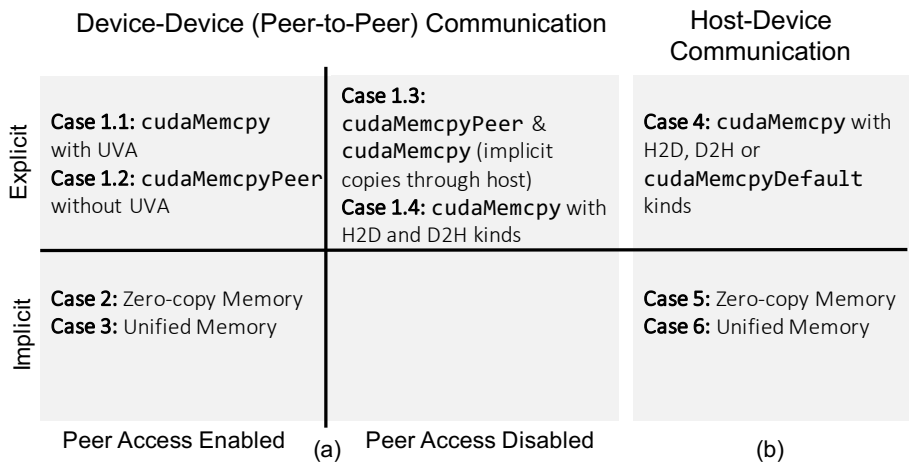


Fig. 1: Types of communication in a multi-GPU system. D2H stands for "Device to Host", H2D stands for "Host to Device".

whether peer access is enabled or disabled and whether the application is using UVA, Zero-copy Memory or Unified Memory.

Peer-to-Peer Explicit Transfers: The host can be explicitly programmed to carry out peer-to-peer transfers using CUDA API. GPU's support for peer access determines the communication pattern in the node, for instance, in a node where GPUs do not have peer access, GPU-GPU communication will happen through the host. Furthermore, the choice of CUDA primitives depend on the utilization of UVA in data transfers. For example, `cudaMemcpyDefault` transfer kind can not be used without UVA.

Case 1: cudaMemcpy and cudaMemcpyPeer. The host can explicitly initiate a peer-to-peer transfer using either `cudaMemcpy` or `cudaMemcpyPeer`. These functions have several cases to consider:

Case 1.1: cudaMemcpy with UVA. UVA enables a single address space for all CPU and GPU memories. It allows determining the physical memory location from the pointer value and simplifies CUDA memory copy functions. For GPUs with peer access and UVA, data transfers between devices are initiated by the host, using `cudaMemcpyDefault` as the transfer kind with `cudaMemcpy`, also known as a UVA memory copy. `cudaMemcpyDefault` is only allowed on systems that support UVA and it infers the direction of transfer from pointer values, e.g. if two pointers point to different devices, then a peer-to-peer memory transfer occurs without specifying in which memory space source and destination pointers are. An alternative to this would be to use `cudaMemcpy` with the transfer kind `cudaMemcpyDeviceToDevice` instead.

Case 1.2: cudaMemcpyPeer without UVA. For CUDA devices that do not support UVA but have peer access enabled, the programmer uses `cudaMemcpyPeer` and explicitly specifies the source device ID where the data to be transferred resides, as well as the destination device ID, in addition to the buffer size, source and destination pointer arguments.

Case 1.3: cudaMemcpy and cudaMemcpyPeer without Peer Access. For GPUs without peer access support or applications that explicitly disable peer access, using `cudaMemcpy` or `cudaMemcpyPeer` will result in implicit transfers through host. CUDA will transparently copy the data from source device to the host and then transfers it to the destination device.

Case 1.4: cudaMemcpy through Host. GPUs that do not have peer-to-peer access or hardware support for Unified Memory require programmer to transfer data among GPUs through the host. `cudaMemcpy` with transfer kind `cudaMemcpyDeviceToHost` is called to copy data from the device to host, and then again with transfer kind `cudaMemcpyHostToDevice` to transfer data from host to another device. This movement of data through host introduces additional overhead, similar to *Case 1.3*.

Peer-to-Peer Implicit Transfers: One can use UVA or Unified Memory to transfer data among GPUs with peer access with very little programming effort. For a kernel, its data can reside in device memory, in another GPU or in system memory. During the kernel execution, CUDA can infer the location of kernel's data and implicitly migrate it to the local device memory if required [22].

Case 2: Zero-Copy Memory. Devices that support UVA and have peer access enabled can access each others' data through the same pointer. During execution, device kernels can implicitly read or write data into the memory of their peer GPUs. Zero-copy Memory transfers occur over the device interconnects without any explicit control by the programmer. The interconnect speed, access pattern and degree of parallelism affect the performance of these transfers [28].

Case 3: Unified Memory. With Unified Memory, data is allocated with `cudaMallocManaged`, which returns a pointer that is accessible from CPU and any GPU in the shared memory node. In this case, if a single pointer is allocated on one device using CUDA managed memory and another device accesses it during kernel execution, then the GPU driver will migrate the page from one device memory to another [14]. Page migrations occur between devices in the event of a page fault, which results in an implicit data transfer.

3.2 Host-Device Communication

Similar to peer-to-peer communication between GPUs, CUDA explicit data transfer primitives, Zero-copy Memory and Unified Memory operations exist for communication between host and device. The following cases describe each option in more detail.

Case 4: cudaMemcpy. Traditionally, after initializing data on the host, the data is transferred to the device using `cudaMemcpy` with the transfer kind of `cudaMemcpyHostToDevice`. Data is transferred from device to host using `cudaMemcpy` with the transfer kind of `cudaMemcpyDeviceToHost`. Using transfer kinds such as `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` mean that the direction of transfer is explicitly specified by the programmer. Or, with the support of UVA, these transfers can happen without explicitly describing the direction by calling `cudaMemcpy` with `cudaMemcpyDefault` as the transfer kind. CUDA will infer the direction of transfer by analyzing the pointer, similar to *Case 1.1*.

Case 5: Zero-Copy Memory. Zero-copy Memory paradigm uses mapped pinned memory that allows a GPU to directly access host memory over PCIe or NVLink interconnect. Page-locked mapped host memory is allocated using `cudaHostAlloc` or `cudaHostRegister` with `cudaHostRegisterMapped`. Next, the device pointer referring to the same memory is acquired with `cudaHostGetDevicePointer` and therefore no explicit data transfers are needed. During kernel execution, the device pointer directly accesses the mapped pinned host memory [24]. However, it must be noted that only unidirectional data transfers that are `read` and `write` operations by the device are possible in this case as the host is unable to perform `write` operations in Zero-copy Memory paradigm [28].

Case 6: Unified Memory. As mentioned in *Case 3*, for Unified Memory, data is allocated via `cudaMallocManaged` which returns a pointer accessible from any processor. If a pointer is allocated on host using `cudaMallocManaged` and the device accesses it during kernel execution, then driver will migrate the page from host to device memory. This holds true for the host as well. In the event of a page fault, page migration occur.

4 Design and Implementation of ComScribe

We have developed ComScribe on top of *nvprof* to identify multi-GPU and CPU-GPU communications. *nvprof* is a light-weight command-line profiler available since CUDA 5.0 [25]. Although NVIDIA's latest profiling tool *Nsight System* is a system-wide performance analysis tool designed to visualize an application's behaviour [27], it does not provide its output in a machine-readable format. On the other hand, *nvprof*'s profiling output can be stored in a text or csv format, which is machine-readable and can be parsed by our tool. Though, *nvprof* profiles data transfers between CPU-GPU and GPU-GPU during the execution of an application, the data required for generating intra-node communication matrices i.e. total amount of data shared between each pair of devices in a node for each type of communication is not readily available, requiring extra effort by the programmer to extract such information. For example, for each kernel or memory copy, detailed information such as kernel parameters, shared memory usage and memory transfer throughput are recorded. As a result, it becomes difficult for the user to observe the total data movement and types of communication between each pair of GPUs in a node. ComScribe represents the communication

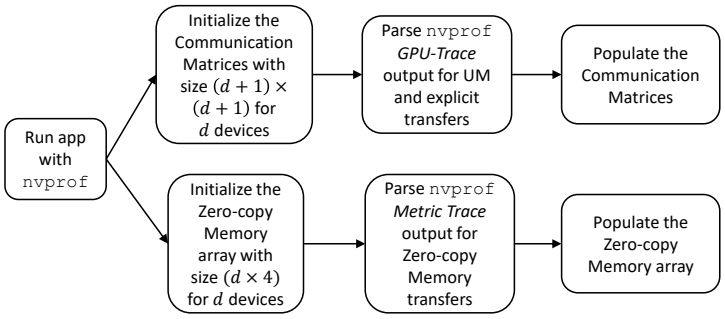


Fig. 2: Workflow diagram of ComScribe

pattern in a compact though descriptive manner to the user, in the light of the data obtained and parsed from *nvprof*.

The *nvprof* profiler has 5 modes namely *Summary*, *GPU-Trace*, *API-Trace*, *Event/Metric Summary* and *Event/Metric Trace*. Among these only GPU-Trace and Event/Metric Trace modes are necessary for our tool. The GPU-Trace mode provides information of all activities taking place on a GPU such as kernel execution, memory copy and memory set and is used by ComScribe for generating communication matrices for explicit and Unified Memory data transfers. For Zero-copy Memory transfers, the amount of data transmitted and received through NVLink by a single GPU in a node is drawn from *nvprof*'s Metric Trace mode. Since *nvprof* operates in one mode at a time and the information for all types of communication is not available through a single mode, ComScribe runs the application twice with *nvprof* to collect necessary information.

ComScribe parses *nvprof*'s output for memory copies, accumulates the communication amount for each GPU-GPU and CPU-GPU pair, infers the types of communication and then generates intra-node communication matrices for an application using CUDA's explicit data transfer primitives or Unified Memory. ComScribe can generate these matrices for both the number of bytes transferred, and the number of data transfers. For Zero-copy Memory, *nvprof's* Metric Trace mode lacks information about the sending or receiving GPU, making it infeasible to generate communication matrix. Instead our tool constructs a bar chart that represents the data transmitted to and received from another GPU or CPU for write and read operations, respectively for each GPU.

4.1 ComScribe Workflow

To generate communication matrices, our tool runs the application code with *nvprof*'s GPU-Trace and Metric Trace modes. The profiling output is parsed only for memory copies by our tool. The type of memory copy helps ComScribe identify the CUDA data transfer primitive. The workflow is shown in Figure 2 and explained below:

1. **Running application with nvprof.** By running the application with *nvprof*'s GPU-Trace and Metric Trace modes, all activities taking place in

the node such as kernel executions and memory copies are recorded in a chronological order.

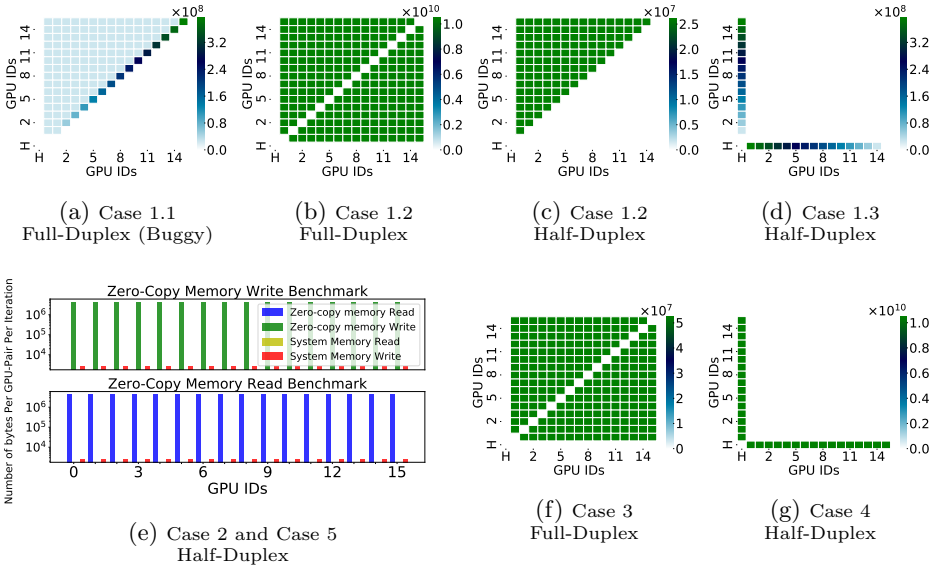
2. **Initialization of communication matrices and Zero-copy Memory array.** Depending on the number of devices, d, available in the node, communication matrices represented as two-dimensional arrays of size $(d+1)\times(d+1)$ are initialized to zero where $+1$ accounts for CPU. Communication matrices are generated for both explicit data transfers and Unified Memory transfers. For Zero-Copy Memory transfers, a one-dimensional array of size $(d \times (4)$ is initialized to recorded data transferred and received through NVLink and system memory. A Zero-copy Memory read or write can be identified for a GPU's memory and host memory, so in total this gives 4 types of transfer for every device.
3. **Recording data transfers.** *nvprof's* GPU-Trace mode output is parsed by ComScribe, which reads each recorded event to detect if it is an event of data transfer between GPU-GPU or CPU-GPU. On each memory copy, the source device ID, destination device ID, size of data transferred is extracted. The size is converted into bytes, then by looking at the source and destination device IDs, the communication event is recorded to the communication matrix, where y-axis indicates receivers and x-axis indicates senders. In this matrix, we can also see the host as a sender or receiver, at the 0^{th} index with the letter H. Data movements within a GPU such as local memory copies are also recorded, which appear as non-zero entries in the diagonal of the communication matrix. Moreover, memory copies are characterized to identify the types of communication. For Zero-copy Memory transfers the Metric Trace mode output is parsed to record data movement for each GPU.
4. **Generating Results.** After the memory copy information are recorded in communication matrices, these matrices are plotted for each identified type of communication, in the case of explicit and Unified Memory transfers. Zero-copy Memory transfers are represented in the form of a bar chart.

5 Evaluation

This section evaluates ComScribe on selected micro-benchmarks from the MGBench and Comm|Scope [7, 28] benchmark suites. We also present communication matrices for macro-benchmarks including NVIDIA's Ising-GPU and Multi-GPU Jacobi Solver, MGBench's Game of Life (GOL) and study two deep learning applications: E3D-LSTM and Transformer [23, 21, 7, 38, 35] as use-cases. The evaluation is conducted on a DGX-2 system consisting of 16 NVIDIA Tesla V100 GPUs, allowing simultaneous communication between 8 GPU pairs at 300 GBps through 12 integrated NVSwitches [20]. We use CUDA v10.0.130.

5.1 Micro-benchmarks from Comm|Scope and MGBench

To validate our tool, we have selected 8 micro-benchmarks with known communication patterns, 6 from Comm|Scope and 2 from MGBench based on their



(a) Case 1.1 Full-Duplex (Buggy)
(b) Case 1.2 Full-Duplex
(c) Case 1.2 Half-Duplex
(d) Case 1.3 Half-Duplex
(e) Case 2 and Case 5 Half-Duplex
(f) Case 3 Full-Duplex
(g) Case 4 Half-Duplex

Fig. 3: ComScribe results on several micro-benchmarks from Comm|Scope (Figures 3a, 3c, 3d, 3e and 3f) and MGBench (Figures 3b and 3g) benchmark suites.

communication types using explicit data transfers with peer access enabled and disabled, as well as implicit data transfers using Unified Memory and Zero-copy Memory operations. We have observed that the communication matrices for number of bytes transferred and number of transfers are very similar in these benchmarks, therefore we only include the former in this section. Each Comm|Scope micro-benchmark is run with 100 iterations and transfers 256KB data. Whereas, MGBench by default runs with 100MB for memory transfers and for a total of 100 iterations.

Figures 3a-3d show the communication matrices generated by ComScribe for the micro-benchmarks that use peer-to-peer explicit transfers. Benchmarks in Figures 3a and 3b use `cudaMemcpyAsync` and `cudaMemcpyPeerAsync` data transfer primitives, respectively for peer-to-peer full-duplex communication with peer access enabled (*Case 1.1 and 1.2*). However, the pattern in the communication matrix in Figure 3a is not as expected– should have been similar to the one in Figure 3b, which shows the data movement as a result of bidirectional transfers between peer GPUs. We observe data transfers in one direction between GPUs along with data movement within the GPU, which translates to local memory copies. With the help of ComScribe, we were able to spot this incorrect communication pattern and identify a communication bug in Comm|Scope's *`cudaMemcpyAsync-duplex`* micro-benchmark on GitHub[1]. After investigating the benchmark code, we realized that the source and destination pointers of the

[1] https://github.com/c3sr/comm_scope/blob/master/src/cudaMemcpyAsync-duplex/gpu_gpu_peer.cpp

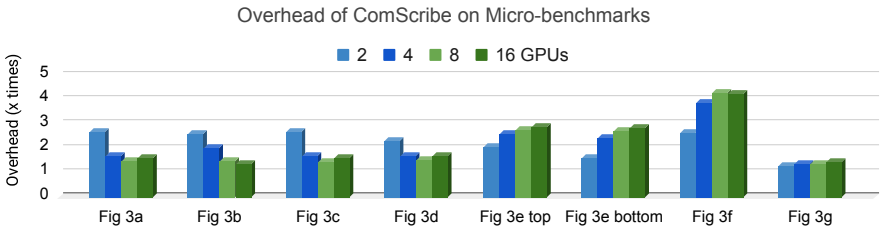


Fig. 4: ComScribe's overhead (mostly stemming from *nvprof*) for Comm|Scope and MGBench micro-benchmarks on 2, 4, 8, and 16 GPUs.

second copy are unintentionally pointing to the same device, resulting in data movement within the device.

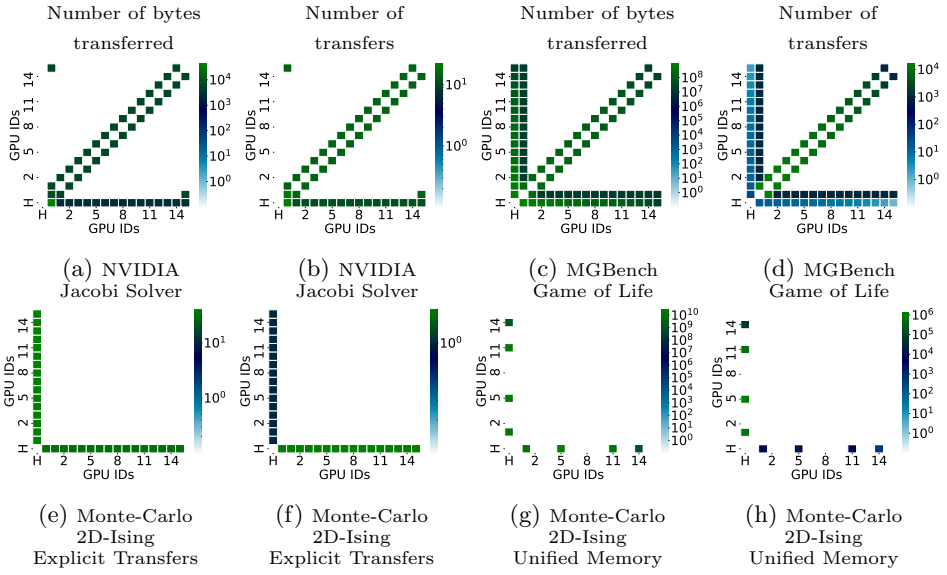
Figures 3c and 3d present results for peer-to-peer half-duplex micro-benchmarks where unidirectional communication between each pair of GPUs is expected. However, the key difference between two is that the benchmark in Figure 3c uses `cudaMemcpyPeerAsync` with peer access (*Case 1.2*) while the benchmark in Figure 3d uses `cudaMemcpyPeerAsync` without peer access (*Case 1.3*). Figure 3c and 3d demonstrate how peer access being enabled or disabled affects the communication pattern. In Figure 3c, the transfers are made directly to the peer GPU, whereas, in Figure 3d, the data is implicitly transferred through host memory as peer access is disabled.

Figures 3e and 3f show results for implicit transfers that occur through Zero-copy Memory and Unified Memory operations (*Case 2, 3 and 5*). Zero-copy Memory bar-chart generated by ComScribe in Figure 3e shows the number of bytes received and transmitted by a single GPU in Comm|Scope Zero-copy Memory GPU-GPU read and write half-duplex micro-benchmarks, respectively, where a small amount of system memory write is detected as well (*Case 5*). The benchmark in Figure 3f uses Unified Memory for peer-to-peer full-duplex communication. We can observe communication between all pairs of GPUs in the node as a result of bidirectional Unified Memory data transfers. However, the total number of bytes transferred among each pair of GPUs is twice than expected due to prefetching i.e. after GPU0 transfers data to GPU1, in the next iteration the data is prefetched from GPU1 to GPU0 before it is sent to GPU2.

Figure 3g illustrates the communication type of *Case 4* with the MGBench's Scatter-Gather micro-benchmark that uses `cudaMemcpyAsync` with transfer kind `cudaMemcpyHostToDevice` to scatter data from host to all GPUs and then gathers data back from GPUs to host using `cudaMemcpyAsync` with transfer kind `cudaMemcpyDeviceToHost`.

5.2 Overhead

Figure 4 presents the overhead of running ComScribe with the micro-benchmarks. *nvprof* is the main contributor to the overhead of ComScribe as post-processing of *nvprof* profiling outputs is negligible. On average, an overhead of 2.17x is observed, due to the fact that ComScribe runs the application with *nvprof* twice,



(a) NVIDIA Jacobi Solver (b) NVIDIA Jacobi Solver (c) MGBench Game of Life (d) MGBench Game of Life

(e) Monte-Carlo 2D-Ising Explicit Transfers (f) Monte-Carlo 2D-Ising Explicit Transfers (g) Monte-Carlo 2D-Ising Unified Memory (h) Monte-Carlo 2D-Ising Unified Memory

Fig. 5: ComScribe results on macro-benchmarks from NVIDIA (Figures 5a, 5b, 5e, 5f, 5g and 5h) and MGBench (Figures 5c and 5d).

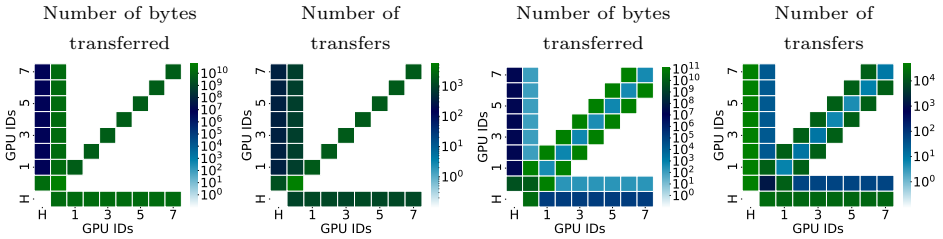
once with GPU-Trace mode for explicit and Unified Memory transfers, and then once more with Metric Trace mode for Zero-copy Memory transfers.

5.3 Macro-Benchmarks from NVIDIA and MGBench

We present insightful communication matrices for three macro-scale applications using 16 GPUs. These applications are NVIDIA's Multi-GPU Jacobi Solver, NVIDIA's Monte Carlo Simulation 2D Ising-GPU, and MGBench's Game of Life [21, 23, 7]. Unlike micro-benchmarks, here we include both matrices: number of bytes transferred and number of transfers. We use log scale instead of linear scale in the figures to make the small transfers visually noticeable.

Figures 5a and 5b show a communication pattern that is prevalent among stencil applications such as Jacobi solver: a GPU shares a "halo region" with it's nearest neighbors and it communicates with them in every iteration to exchange and update the halo cells. This access pattern can be thought of as a "window" [6] and shows itself as non-zero entries on the diagonal in the communication matrix. Note that this application uses circular boundary condition where every cell including the ones at the boundary has a neighbor in all directions, which presents itself as communication between GPU0 and GPU15.

Game of Life, shown in Figures 5c and 5d, is also a type of stencil application. It's domain is distributed among GPUs, and the state of the domain in the next iteration is computed by every GPU using the current state. Similarly, there is a halo region that needs to be updated every iteration, and the nearest-neighbor communication reflects itself as non-zero entries in the diagonal of the



(a) Eidetic 3D LSTM (b) Eidetic 3D LSTM (c) Transformer (d) Transformer

Fig. 6: ComScribe results on two Deep Neural Network models implemented in TensorFlow framework using data parallelism across 8 GPUs

communication matrix. Unlike Jacobi, this application does not employ a circular boundary condition. Note that the communication values are not equal among the GPUs because this benchmark runs the application in increments, starting from 1 GPU, increasing one at a time until the desired amount of GPUs, 16 in our case, is reached. As a result, GPU0 is used in all 16 runs, GPU1 is used in 15 runs, and so on. We can observe this at the diagonals of both figures and from the L shape pattern in Figure 5c in which the number of bytes transferred and number of transfers decrease as as the GPU IDs increase.

Figures 5e-5h shows the results of GPU-accelerated Monte Carlo simulations of 2D Ising Model. The algorithm uses a scatter-gather pattern where independent simulations are run on each GPU and then they are gathered at the host for averaging. Scatter-gather is distinguishable as it forms an L shape pattern on the communication matrix, where the host sends data to all GPUs, and receives data from all. We observe that this application uses both explicit transfers and Unified Memory at the same time, which is identified by ComScribe. By looking at the explicit transfer's communication matrices, we can immediately see the intended communication pattern, which is also complied by the Unified Memory transfers. Moreover, this application (Figures 5g and 5h) serves as an example for the type of communication presented as *Case 6*.

5.4 Use-Cases: Deep Neural Network (DNN) Models

This section demonstrates the communication matrices for two DNN models. GPUs are commonly used by the deep learning frameworks to accelerate both the training and inference. Due to its simplicity and desired weak scaling, the dominant distributed DNN training strategy is to use data parallelism, where input is split among multiple GPUs each of which holds a replica of the model [34, 30]. In stochastic gradient descent (SGD) [2], where the training is done in mini-batches, the weights of the DNN need to be updated. In the update phase, the results of the replicas have to be averaged to obtain the gradient of the whole minibatch [5]. The averaged version then needs to be accessed by all the replicas to update their parameters. The pervasiveness of data parallelism has resulted in having multiple underlying implementations (communication patterns) for the gradient exchange. Each of these pattern's performance, and hence the training

performance, vary depending on the underlying network topology. By visualizing the communication, our tool helps the user to understand the underlying communication pattern and analyze its effect on performance.

Figures 6 shows communication matrices generated by ComScribe for two well-known deep learning models implemented in the TensorFlow framework [1] using data parallelism to execute on 8 GPUs. The first model is Eidetic 3D LSTM (E3D-LSTM), a spatio-temporal predictive learning model for video prediction [37]. The second is Transformer [36], a widely used model that has considerably influenced the design of SoTA Transformer based models in the NLP domain. From the communication matrices of these two models, two variations of communication used for implementing data parallelism can be observed. In the E3D-LSTM case, the gradients are sent from GPUs to host, they are averaged, then sent from host to GPU0. After this, GPU0 scatters the gradients to all other GPUs. For Transformer, the gradients are exchanged using the nearest neighbor communication, as a result the largest amount of communication occurs between neighboring pairs of GPUs. While in E3D-LSTM, the second largest communication is self communication happening due to the huge memory allocated temporarily for 3D-convolutions. In Transformer, it occurs between GPU0 and host for the purpose of training checkpoint, where a copy of the model state and its parameters are sent to the host. The remaining communication takes place between the host and devices to distribute the training samples in the batch across the GPUs. Note that the models we used in our demonstration are large models having O(1 billion) parameters that is why the last type of communication is relatively small. However, it might have higher relative importance for smaller models.

6 Related Work

EZTrace is a generic trace generation framework for programs that are written using OpenMP, MPI, PThreads and CUDA [33]. It generates trace for such programs in two steps: first, it collects the necessary information by intercepting function calls and recording events during program execution using the FxT library [10], then it performs a post-mortem analysis on the recorded events. The total message size exchanged and number of messages for peer-to-peer and collective communications can be generated for MPI with the help of EZTrace, but for CUDA this feature is limited. For CUDA applications, a log of recorded events is generated that includes the total message size exchanged and number of messages for CPU-GPU communication that uses `cudaMalloc-cudaMemcpy` pair only. In comparison, our tool detects a much diverse set of communication types and generates communication matrices for GPU-GPU communication including all CUDA explicit data transfer primitives, Zero-copy Memory and Unified Memory operations.

Comm|Scope [28] is a set of micro-benchmarks to measure latency and bandwidth for CUDA data transfer primitives with different data placement scenarios. This includes NUMA-aware point-to-point CPU-GPU and GPU-GPU explicit data transfer primitives in CUDA as well as Zero-copy Memory and Unified

Memory operations. Although our work and Comm|Scope are parallel in motivation to categorize types of communication, Comm|Scope does not focus on identifying and quantifying GPU communications. NUMA-aware communication affects the performance, however it does not change the amount of data shared among GPUs, thus it is not included in our work. Moreover, we consider additional CUDA data transfer primitives like `cudaMemcpyPeer` and transfer kind `cudaMemcpyDefault`.

Li et. al [16, 15] developed Tartan, a multi-GPU benchmark suite consisting of micro-benchmarks for characterizing GPU interconnects and 14 application benchmarks for evaluating how multi-GPU execution performance is affected by different GPU interconnects. Tartan includes micro-benchmarks to evaluate interconnects including PCIe, NVLink 1.0, NVLink 2.0, NV-SLI, NVSwitch and Infiniband systems with GPUDirect RDMA for point-to-point and collective intra-node and inter-node GPU-GPU communication. It measures communication efficiency, bandwidth and latency of GPU-GPU explicit memory copies and the NVIDIA Collective Communications Library (NCCL) on the interconnects. However, Tartan is not a tool but rather a benchmark suite, whereas, we offer a tool that captures the total amount of data transferred for intra-node peer-to-peer communication.

A number of tools that generate communication patterns for multi-core applications exist in the literature. ComDetective [29] is a sampling based tool that uses Performance Monitoring Units (PMUs) and debug registers to detect inter-thread data transfers and generates communication matrices for multi-threaded applications for shared memory systems. Likewise, Azimi et al. [3] and Tam et al. [32] leverage kernel support to access PMUs and the kernel generates the communication pattern for the applications. Barrow-Williams et al. [4] and Cruz et al. [9] employ simulator-based approach to collect memory access traces for generating communication patterns. Lastly, Numalize [12, 11] uses binary instrumentation to intercept memory accesses and captures communication between threads accessing the same address in memory. However, none of these tools can be used to identify multi-GPU communication.

7 Conclusion

In single node multi-GPU systems, communication is a critical programming component and performance contributor. For such systems, we categorize various data transfer options offered by CUDA API to the programmer, into explicit data transfer primitives and implicit transfers using Unified Memory and Zero-copy Memory operations for Peer-to-Peer and Host-Device communications. We developed ComScribe on top of NVIDIA's profiling tool *nvprof* that identifies, quantifies and generates communication matrices for GPU-GPU and CPU-GPU communications in a node. ComScribe's workflow is presented, followed by it's evaluation on several micro- and macro-benchmarks as well as two deep learning applications. Communication matrices generated by our tool can be used by programmers to differentiate types of communication, study the communication patterns and detect communication bugs in a multi-GPU application.

Acknowledgement

Some of the authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 118E801. The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 265–283. OSDI'16, USENIX Association, USA (2016)
2. Amari, S.i.: Backpropagation and stochastic gradient descent method. Neurocomputing **5**(4-5), 185–196 (1993)
3. Azimi, R., Tam, D.K., Soares, L., Stumm, M.: Enhancing operating system support for multicore processors by using hardware performance monitoring. ACM SIGOPS Operating Systems Review **43**(2), 56–65 (2009)
4. Barrow-Williams, N., Fensch, C., Moore, S.: A communication characterisation of splash-2 and parsec. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 86–97. IEEE (2009)
5. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) **52**(4), 1–43 (2019)
6. Ben-Nun, T., Levy, E., Barak, A., Rubin, E.: Memory access patterns: The missing piece of the multi-GPU puzzle. International Conference for High Performance Computing, Networking, Storage and Analysis, SC **15-20-November-2015** (2015). https://doi.org/10.1145/2807591.2807611
7. Ben-Nuun, T.: Mgbench: Multi-gpu computing benchmark suite (cuda). https://github.com/tbennun/mgbench (2017), (Accessed on 07/29/2020)
8. Buono, D., Artico, F., Checconi, F., Choi, J.W., Que, X., Schneidenbach, L.: Data analytics with NVLink: An SpMV case study. ACM International Conference on Computing Frontiers 2017, CF 2017 pp. 89–96 (2017). https://doi.org/10.1145/3075564.3075569
9. da Cruz, E.H.M., Alves, M.A.Z., Carissimi, A., Navaux, P.O.A., Ribeiro, C.P., Méhaut, J.F.: Using memory access traces to map threads and data on hierarchical multi-core platforms. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. pp. 551–558. IEEE (2011)
10. Danjean, V., Namyst, R., Wacrenier, P.A.: An efficient multi-level trace toolkit for multi-threaded applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005 Parallel Processing. pp. 166–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
11. Diener, M., Cruz, E.H., Alves, M.A., Navaux, P.O.: Communication in shared memory: Concepts, definitions, and efficient detection. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 151–158. IEEE (2016)

12. Diener, M., Cruz, E.H., Pilla, L.L., Dupros, F., Navaux, P.O.: Characterizing communication and page usage of parallel applications for thread and data mapping. Performance Evaluation **88**, 18–36 (2015)
13. Foley, D., Danskin, J.: Ultra-performance pascal gpu and nvlink interconnect. IEEE Micro **37**(2), 7–17 (2017)
14. Harris, M.: Unified memory for cuda beginners — nvidia developer blog. https://devblogs.nvidia.com/unified-memory-cuda-beginners/ (June 2017)
15. Li, A., Song, S.L., Chen, J., Li, J., Liu, X., Tallent, N.R., Barker, K.J.: Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. IEEE Transactions on Parallel and Distributed Systems **31**(1), 94–110 (2020)
16. Li, A., Song, S.L., Chen, J., Liu, X., Tallent, N., Barker, K.: Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In: 2018 IEEE International Symposium on Workload Characterization (IISWC). pp. 191–202 (2018)
17. Micikevicius, P.: Multi-gpu programming. https://on-demand.gputechconf.com/gtc/2012/presentations/S0515-GTC2012-Multi-GPU-Programming.pdf (2012)
18. NVIDIA: Nvidia gpudirect technology. http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf (2012)
19. NVIDIA: Nvidia dgx-1 with tesla v100 system architecture white paper. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf (2017), (Accessed on 07/28/2020)
20. NVIDIA: Dgx-2 : Ai servers for solving complex ai challenges — nvidia. https://www.nvidia.com/en-us/data-center/dgx-2/ (2018), (Accessed on 07/28/2020)
21. NVIDIA: Multi-gpu-programming-models: Examples demonstrating available options to program multiple gpus in a single node or a cluster. https://github.com/NVIDIA/multi-gpu-programming-models (2018), (Accessed on 07/29/2020)
22. NVIDIA: Cuda runtime api. https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf (July 2019), (Accessed on 07/29/2020)
23. NVIDIA: Ising-gpu: Gpu-accelerated monte carlo simulations of 2d ising model. https://github.com/NVIDIA/ising-gpu (2019), (Accessed on 07/29/2020)
24. NVIDIA: Best practices guide :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#zero-copy (2020), (Accessed on 05/12/2020)
25. NVIDIA: Cuda profiler user's guide. https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf (July 2020), (Accessed on 07/28/2020)
26. NVIDIA: Dgx a100 : Universal system for ai infrastructure — nvidia. https://www.nvidia.com/en-us/data-center/dgx-a100/ (2020), (Accessed on 07/28/2020)
27. NVIDIA: Nvidia nsight systems documentation. https://docs.nvidia.com/nsight-systems/index.html (2020), (Accessed on 07/28/2020)
28. Pearson, C., Dakkak, A., Hashash, S., Li, C., Chung, I.H., Xiong, J., Hwu, W.M.: Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. p. 209–218. ICPE '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297663.3310299, https://doi.org/10.1145/3297663.3310299
29. Sasongko, M.A., Chabbi, M., Akhtar, P., Unat, D.: Comdetective: a lightweight communication detection tool for threads. In: Proceedings of the International

Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–21 (2019)
30. Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al.: Mesh-tensorflow: Deep learning for supercomputers. In: Advances in Neural Information Processing Systems. pp. 10414–10423 (2018)
31. Sourouri, M., Gillberg, T., Baden, S.B., Cai, X.: Effective multi-GPU communication using multiple CUDA streams and threads. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS **2015-April**(1), 981–986 (2014). https://doi.org/10.1109/PADSW.2014.7097919
32. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. ACM SIGOPS Operating Systems Review **41**(3), 47–58 (2007)
33. Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., Dongarra, J.: Eztrace: a generic framework for performance analysis. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 618–619. IEEE (2011)
34. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (1990)
35. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. CoRR **abs/1706.03762** (2017), http://arxiv.org/abs/1706.03762
36. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in neural information processing systems. pp. 5998–6008 (2017)
37. Wang, Y., Jiang, L., Yang, M.H., Li, L.J., Long, M., Fei-Fei, L.: Eidetic 3d lstm: A model for video prediction and beyond. In: International Conference on Learning Representations (2018)
38. Wang, Y., Jiang, L., Yang, M.H., Li, L.J., Long, M., Fei-Fei, L.: Eidetic 3d LSTM: A model for video prediction and beyond. In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=B1lKS2AqtX