GPU-Centric Communication Schemes: When CPUs Take a Back Seat

by

Ismayil Ismayilov

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

August 10, 2023

GPU-Centric Communication Schemes: When CPUs Take a Back Seat

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Ismayil Ismayilov

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Dr. Didem Unat (Advisor)

_____

Assoc. Prof. Dr. Alptekin Küpçü

_____

Prof. Dr. Can Özturan

Date: _____

ABSTRACT

GPU-Centric Communication Schemes: When CPUs Take a Back Seat
Ismayil Ismayilov
Master of Science in Computer Science and Engineering
August 10, 2023

In recent years, GPUs have become the leading accelerator in modern high-performance systems such that much of HPC computational capability has concentrated in clusters of GPUs. Using multi-GPU acceleration has brought great computational benefits to many HPC and ML applications. However, the need to communicate between GPUs, both within and across nodes, can quickly become a bottleneck that hinders application scaling. A significant reason for this is that traditionally communication has been mediated through the host. In a typical multi-GPU application, the host orchestrates execution by launching kernels, issuing communication calls, and acting as a synchronizer for devices. This CPU involvement in the critical path of execution causes undue overhead and can be delegated entirely to devices to improve performance in applications that involve multi-GPU communication.

We first present a fully autonomous execution model for single-node multi-GPU applications that completely excludes the involvement of the CPU beyond the initial kernel launch. For the proposed *CPU-free* execution model, we leverage existing techniques such as persistent kernels, thread block specialization, device-side barriers, and device-initiated communication routines to write fully autonomous multi-GPU code and achieve significantly reduced communication overheads. We demonstrate our proposed model on two variants of the broadly used Conjugate Gradient (CG) solver, Standard CG, and Pipelined CG. Compared to the CPU-controlled baselines, the CPU-free model provides a 1.54x and 1.63x speedup for Standard and Pipelined CG, respectively, on 8 NVIDIA A100 GPUs.

In the second part of the thesis, we conduct an extensive survey of GPU-centric communication, communication mechanisms proposed in response to the deficiencies of traditional multi-GPU communication models. At a high level, these advancements reduce the CPU's involvement in the critical path of execution, give the GPU more autonomy in initiating and synchronizing communication and fix the seman-

tic mismatch between multi-GPU communication and computation. We chart out the landscape of GPU-centric communication, summarize the main methods and expound on their most salient features, including associated benefits and challenges.

ÖZETÇE

GPU-Odaklı Haberleşme Sistemleri: CPU'ların Arka Koltuğa Geçtiği Zamanlar

Ismayil Ismayilov

Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans

10 Ağustos 2023

Son yıllarda GPU'lar, modern yüksek performanslı sistemlerde önde gelen hızlandırıcı haline gelmiştir ve bu nedenle HPC hesaplama gücünün büyük bir kısmı GPU kümelemelerine odaklanmıştır. Çoklu GPU hızlandırma kullanımı, birçok HPC ve Makine Öğrenmesi uygulamasına büyük hesaplama avantajları getirmiştir. Ancak GPU'lar arasında, hem düğümler içinde hem de aralarında iletişim kurma ihtiyacı, uygulama ölçeklendirmesini engelleyen bir darboğaz haline gelebilir. Bunun önemli bir nedeni, geleneksel olarak iletişimin ana bilgisayar üzerinden yönetilmesidir. Tipik bir çoklu GPU uygulamasında ana bilgisayar, çekirdekleri başlatarak, iletişim çağrıları yaparak ve cihazlar için bir senkronizasyon sağlayarak yürütümü yönetir. Bu, yürütümün kritik yolunda CPU'nun dahil olması, gereksiz bir iş yükü oluşturur ve çoklu GPU iletişimi içeren uygulamalarda performansı artırmak için cihazlara tamamen devredilebilir.

İlk olarak, tek düğümlü çoklu GPU uygulamaları için tamamen otonom bir yürütüm modeli sunuyoruz, bu da başlangıçta çekirdek başlatma dışında CPU'nun dahil edilmediği anlamına gelir. Önerilen *CPU'suz* yürütüm modelinde, mevcut teknikleri, kalıcı çekirdekler, iş parça özelleştirme, cihaz tarafından başlatılan bariyerler ve cihaz tarafından başlatılan haberleşme çağrıları gibi teknikleri kullanarak tamamen otonom çoklu GPU kodu yazmak ve iletişim üzerinde önemli ölçüde azaltılmış bir iş yükü sağlamak için kullanıyoruz. Önerilen modelimizi, geniş kullanıma sahip iki farklı türe sahip Conjugate Gradient (CG) çözücüsünün, Standart CG ve Pipelined CG'nin üzerinde gösteriyoruz. CPU tarafından kontrol edilen yöntemlerle karşılaştırıldığında, CPU'suz model, 8 NVIDIA A100 GPU'sunda Standart CG ve Pipelined CG için sırasıyla 1.54x ve 1.63x hızlanma sağlar.

Tezin ikinci kısmında, geleneksel çoklu GPU iletişim modellerinin eksikliklerine yanıt olarak önerilen GPU-odaklı iletişimi kapsamlı bir şekilde incelemekteyiz.

Genel olarak, bu ilerlemeler, yürütümün kritik yolundaki CPU'nun dahilini azaltmakta, GPU'ya iletişimi başlatma ve senkronize etme konusunda daha fazla özerklik sağlamakta ve çoklu GPU iletişimi ile hesaplama arasındaki anlamsal uyumsuzluğu gidermektedir. Bu tezde GPU-odaklı iletişimi sınıflandırıyor, temel yöntemleri özetliyor ve faydaları ve zorlukları da içeren en önemli özellikleri üzerinde duruyoruz.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

API	Application Programming Interface
GPU	Graphics Processing Unit
ROCm	Radeon Open Compute platform
DMA	Direct Memory Access
UVA	Unified Virtual Addressing
P2P	Point-to-Point
UVM	Unified Virtual Memory
IPC	Inter-Process Communication
RDMA	Remote Direct Memory Access
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communication Library
RCCL	ROCm Collective Communication Library
PCIe	Peripheral Component Interconnect Express
CG	Conjugate Gradient
TB	Thread Block
UCX	Unified Communication X

Chapter 1

INTRODUCTION

GPUs have become the leading accelerator in modern HPC systems, equipping 7 of the 10 leading Top500 supercomputers in the world [Meuer et al., 2023]. As real-life applications rely on multiple GPUs to solve large problems, communication among GPUs can quickly become a performance bottleneck, leading to poor application scaling [Shimokawabe et al., 2011]. In the traditional model of execution, communication among GPUs in large systems has been mediated through the host. In the absence of direct peer-to-peer communication, GPU-to-GPU data transfers had to be routed through the CPU, which incurred larger communication latencies [Agostini et al., 2017, Hamidouche et al., 2015]. In response to the deficiencies of the traditional model of execution, several advances broadly known as *GPU-centric* communication have been proposed. These solutions span a wide spectrum of approaches and include hardware innovations like proprietary GPU-to-GPU interconnects and software mechanisms like GPU-aware MPI. At a high level, these advancements seek to reduce the CPU's involvement in the critical path of execution, give the GPU more autonomy in initiating and synchronizing communication and fix the semantic mismatch between multi-GPU computation and communication.

While, by the advent of *GPU-centric* communication, data movement can now be delegated to the GPU, the communication control flow still sits firmly on the CPU. Even with direct GPU-to-GPU communication, data transfers are mostly *initiated* from the CPU using host-side API calls, putting it in charge of the overall flow of multi-GPU execution; the CPU enqueues both computational kernels and communication calls to devices, overlaps them if possible, and synchronizes them for functional correctness. The CPU also acts as a global barrier for synchronizing

multiple devices when required. Regardless of whether the communication happens directly between devices, the presence of the CPU in the control path is implicitly assumed.

This work argues that freeing the GPU from the host by moving the control flow to devices has several benefits for multi-GPU applications, particularly in communication-bound settings. To illustrate these advantages, we propose a distinct *CPU-Free* execution model that removes the CPU from said control path and gives autonomy to GPUs to control their communication and synchronization. In order to realize this, we leverage and combine four essential programming concepts. The first technique is to switch from kernels launched sequentially by the host to a long-running persistent kernel [Zhang et al., 2023, Steinberger et al., 2014]. Second, we make use of thread block specialization within our persistent kernel to explicitly overlap communication and computation by reserving a number of blocks for each phase. Third, we adopt *GPU-initiated* communication methods to stage data movement independently of the CPU. Finally, we hand over the task of multi-GPU synchronization to the devices themselves and allow them to synchronize with their peers independently. These changes make GPUs less dependent on the host, allow for more asynchrony, reduce communication latencies, and lead to better communication/computation overlap.

We demonstrate the effectiveness of our execution model on the Standard and Pipelined CG variants of the broadly used Conjugate Gradient (CG) solver. The *CPU-free* execution model is a good fit for iterative solvers like CG as their iterative nature requires communication and computation at each iteration - tasks otherwise managed by the CPU.

After discussing our *CPU-Free* model, we broaden our scope and conduct an extensive survey of GPU-centric communication, communication mechanisms proposed in response to the deficiencies of traditional multi-GPU communication models. We chart out the landscape of GPU-centric communication, summarize the main methods and expound on their most salient features, including associated benefits and challenges. Namely, we taxonomize *GPU-centric* communication techniques, discuss vendor-provided mechanisms and elaborate on how these mechanisms give

rise to larger GPU-centric communication paradigms.

Overall, this thesis makes the following contributions:

- We describe a CPU-free execution model that removes the host from the critical path, grants autonomy to devices, reduces latencies, and achieves better communication/computation overlap.

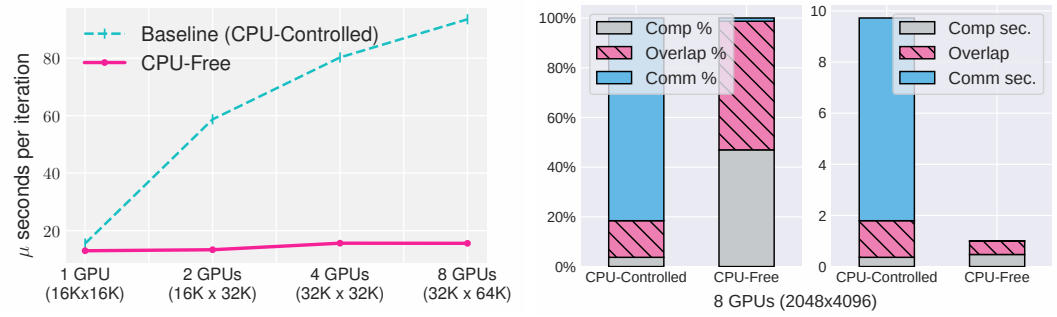
- We design and implement Standard and Pipelined Conjugate Gradient solvers on multiple GPUs using our CPU-free execution model.

- We evaluate the CPU-free execution model and compare its performance against CPU-controlled baselines using 8 NVIDIA A100 GPUs. Standard and Pipelined CG variants observe 1.54x and 1.63x speedup on 18 sparse matrices compared to their CPU-controlled counterparts.

- We conduct an extensive survey of existing GPU-centric communication methods. We discuss the vendor-provided mechanisms that reduce CPU involvement in multi-GPU execution, how higher-level paradigms use those mechanisms to construct GPU-centric communication libraries and expound on applications that use those libraries to get performance benefits.

Chapter 2

BACKGROUND & MOTIVATION



(a) (b)

Figure 2.1: **(a)** Communication and synch overheads with no computation of our CPU-free model against CPU-controlled
(b) Left shows communication overlap percentages for CPU-free and CPU-controlled models. Right shows the execution times.

In this work, we seek to give devices full autonomy by decoupling them from the CPU during multi-GPU execution. We first note that by the advent of GPU-initiated communication [Potluri et al., 2013a, Hamidouche et al., 2015, Potluri et al., 2017], the *data path* - the path in which data is transferred through communication routines - can be moved entirely to the GPU. CUDA devices can perform data transfers among one another through either NVLink [Foley and Danskin, 2017] or PCIe within a node or through NICs in multi-node systems, both avoiding intermediate host-side transfers.

Despite the progress made in pushing the data path to the GPU, overall control of execution - the *control path* - has stayed resident on the host. First, in single-GPU applications, the CPU serves as a barrier that synchronizes kernels with each other. This synchronization is necessary in the case of any iterative solver; to ensure correctness, the computations for timestep $T + 1$ can start only after timestep T

concludes. The CPU satisfies this requirement by launching a kernel every iteration since kernels launched back-to-back are guaranteed to execute serially by the GPU scheduler. This way, the kernel launch and teardown act as an *implicit* barrier between timesteps. For the rest of this work, we refer to such implicitly-synchronized kernels as *discrete* kernels.

For multi-GPU applications, the CPU plays a similarly active role. In addition to kernel launches, the CPU also issues the communication using host-side APIs (i.e., '`cudaMemcpy`'). These calls are issued on the CPU even when the underlying transmission uses the direct GPU-to-GPU data path. When applications allow for overlap between independent phases of communication and computation, it is again the CPU that orchestrates this overlap by enqueueing communication and computation on separate GPU streams to run concurrently and then synchronize through GPU events. Moreover, the CPU acts as a global barrier when synchronizing multiple GPUs, possibly using CPU-side barriers.

```
// ❶ Time loop on CPU
while (iter++ < num_iterations) {
    // ❷ Launch compute kernel
    stencil_kernel<<<..., comp_stream>>>(...)
    // ❸ Sync with neighboring GPUs
    wait_neighbors(comm_stream)
    // ❹ Compute and communicate boundaries
    compute_boundaries<<<..., comm_stream>>>(...)
    write_neighbors(comm_stream)
    // ❺ Sync comm and comp streams
    sync(comm_stream, comp_stream)
}
```

Listing 2.1: Pseudo-code of CPU-controlled stencil solver

Listing 2.1 illustrates the CPU's involvement in the control path with a stencil solver using the *CPU-controlled* model of execution. ❶ First, the CPU maintains a time loop that invokes the computational kernel, communication, and synchronization calls every iteration. ❷ The CPU launches the compute kernel in a `comp_stream`, which performs the stencil operations. ❸ The CPU synchronizes with neigh-

bors to ensure that inbound halos have been received. ❹ Next, the CPU launches a kernel to compute the boundary rows and communicates them as the neighbors' halos. Both of these routines are enqueued to a different stream, namely `comm_stream`, to achieve overlap between communication and computation. Thus, step ❷ is overlapped with step ❸ and ❹. ❺ Finally, the CPU synchronizes these two streams before advancing to the next iteration.

Figure 2.1a presents the communication overheads with no computation - both synchronization and data movement across GPUs - of CPU-controlled and CPU-free executions in increasing problem sizes for the 2D Jacobi solver. The CPU-controlled baseline uses the overlapping implementation shown in Listing 2.1, utilizing host-side CUDA events and memory copy calls to explicitly overlap communication and computation. The CPU-free version instead uses GPU-initiated calls for both of those routines. We notice significantly and consistently lower communication overheads in the CPU-free version in all domain sizes - up to 5.9x at 8 GPUs.

Figure 2.1b shows the ratio of computation and communication overlap of both versions, along with leftover communication overhead that could not be overlapped. We note the overflown communication time in the baseline, where it takes over 96% of the execution, of which only 19% is overlapped with computation, leading to suboptimal performance. We reclaim 89.7% of the full execution time by reducing the communication latency 17.5 times. Though high communication latency can be hidden easily in large domains when computation time dwarfs it, CPU-free execution is particularly of high importance in strong scaling cases and also in phases in simulations where the workload drops (for example, computing boundary conditions for 2D planes in a 3D discretized domain to solve PDEs [Wahib and Maruyama, 2014, Yamaguchi et al., 2017, Afanasyev et al., 2021]).

Chapter 3

CPU-FREE EXECUTION MODEL

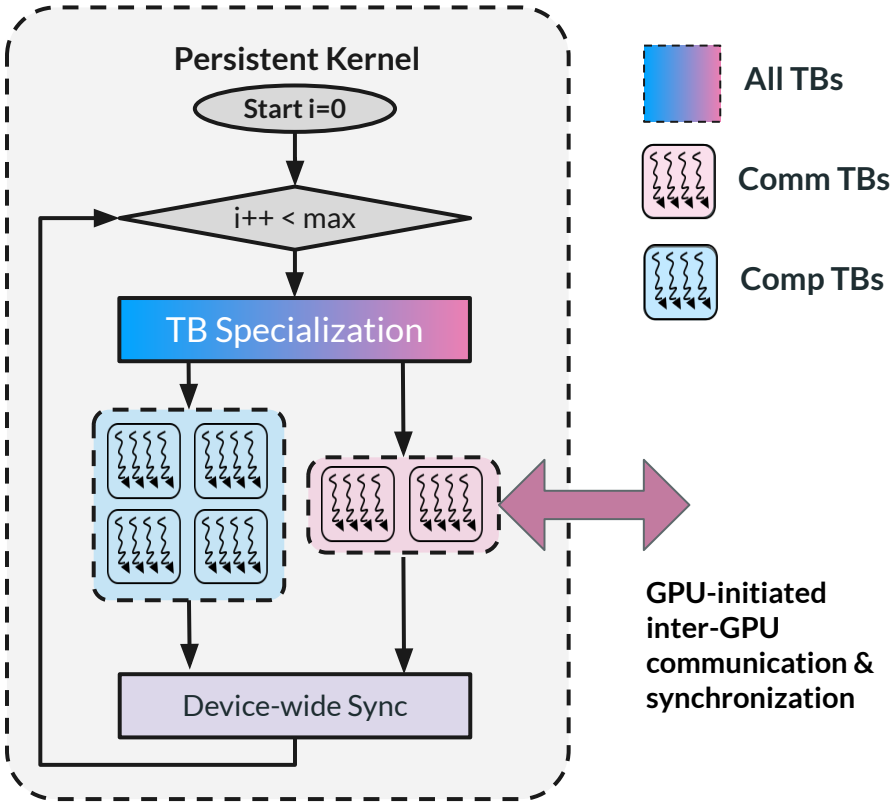


Figure 3.1: CPU-free execution model leverages persistent kernels, thread-block (TB) specialization, GPU-initiated data transfers, and device-side synchronization. Comm: Communication, Comp: Computation.

3.1 Overview

We propose an execution model that eliminates the CPU from both the data and control paths to achieve performance benefits. In order to give full autonomy to GPUs, we make use of several prerequisites and techniques, such as persistent kernels, thread-block specialization, GPU-initiated data transfers, and device-side syn-

chronization. Figure 3.1 shows the concepts that constitute the basis of the CPU-free execution model.

3.1.1 Persistent Kernels

Traditionally, iterative GPU kernels are implemented on a per-iteration basis, meaning a new instance of the kernel is enqueued in a stream for each time step. The GPU is oblivious to the iterative nature of the computation and supplementary operations. This kind of execution relies on the CPU to provide implicit device synchronization across iterations to ensure correctness, as most iterative solvers have temporal dependencies on preceding timesteps. To eliminate such CPU dependence, we implement a long-running persistent kernel, originally proposed in [Gupta et al., 2012], into which we move the time loop of the solver. This allows us to grant more autonomy to the GPU by freeing it from needing to return to the CPU each iteration.

3.1.2 Thread Block Specialization

Key in our design is the nascent idea of *thread block specialization* whereby TBs may work on different tasks within the same kernel (conceptually similar to warp specialization [Bauer et al., 2014]). We utilize this idea to achieve communication/computation overlap to reduce communication latency. Discrete kernels achieve overlap by enqueueing communication and computation kernels in concurrent streams through host-side runtime calls and transferring data with asynchronous `Memcpy` calls that run independently in copy engines. We accomplish this asynchronous behavior on the device side by specializing a number of TBs within a kernel to manage communication while the remainder of the TBs handle the bulk of the computation.

3.1.3 GPU Initiated Data Movement

We use direct GPU-to-GPU data transfers *within* the kernel and initiate GPU communication among peers without host involvement. Direct GPU-to-GPU data movement is used in two ways: to communicate the actual data and to synchronize neigh-

boring GPUs. We use NVSHMEM [Potluri et al., 2017, NVIDIA, 2023h], NVIDIA's implementation of OpenSHMEM [Poole et al., 2011], for all direct GPU-initiated data transfers, as it provides a fine-grained device-side API for data movement.

3.1.4 Device-Side Synchronization

Synchronization in a traditional CUDA kernel is limited to threads within a single thread block, and the kernel launch itself acts as a barrier. Instead, we use device-wide barriers to synchronize the thread blocks across the persistent kernel. At the end of each iteration, device-wide barriers introduced in CUDA 9.0 with the Cooperative Groups API are used to synchronize the communication and computation TBs. Although the latency difference between implicit synchronization using sequential kernel launches and explicit synchronization within the kernel through grid sync is negligible [Zhang et al., 2020], it is no longer required for the CPU to orchestrate the kernel launches just to synchronize threads within a kernel. For synchronization between peer GPUs, traditionally, host-side methods, such as OpenMP and MPI barriers, are used. We move this control back to the GPU using NVSHMEM's device-side signaling operations for peer device synchronization.

3.2 Benefits of CPU-free Execution

As a promising alternative to traditional accelerator programming, the CPU-free execution model has the following benefits over CPU-controlled multi-GPU programming.

1. **Reduced kernel launch/CPU synchronization overheads.** Traditional implementations require multiple API calls to overlap communication and computation. These operations would be launched in separate streams to run concurrently and synchronized through the host. The CPU-free model allows for greater degrees of kernel fusion as the communication and computation can be put inside one fused kernel. This is beneficial because **(i)** one fat kernel substantially reduces the kernel launch overheads

2. **Reduced communication overheads**. We reduce the communication overheads by initiating communication on the device side. Since communication calls can be issued on the device side within the kernel at any point by the GPU as soon as data is ready, it provides more asynchrony by allowing the programmer to inline the communication with the computation. In addition, we reduce cross-GPU synchronization latency by using device-side signaling operations

3. **Communication/Computation overlap**. In traditional implementations, adequate communication/computation overlap can only be achieved when the domain size saturates the device (all threads are busy). But as the problem size per GPU decreases, the kernel launch overheads and the API call latencies can start to dominate the runtime. Because of this, little to no overlap is achieved as the GPU spends a significant amount of time idle. In line with prior work, [Chu et al., 2019], we observe that persistent kernels are able to achieve good overlap even when the domain size is small

4. **Caching.** We extend the lifetime of a kernel by moving the time loop from the host to the device. This enables the use of the large volume of shared memory, registers to cache intermediate results, and prevents wiping out on-chip memory. As demonstrated by the work by [Zhang et al., 2023], this is particularly beneficial for iterative kernels with temporal dependencies between iterations, as the cache would otherwise be destroyed at each time step in a discrete kernel.

3.3 CPU-Free Conjugate Gradient

We apply our CPU-free execution model on top of the Conjugate Gradient (CG) solver, an iterative method used to solve linear equations of the form $Ax = b$ where A is a symmetric and positive definite matrix.

Three primary operations underlie the CG solver: Sparse matrix-vector multiplication (SpMV), dot product, and Saxpy. Assuming vectors are equally divided

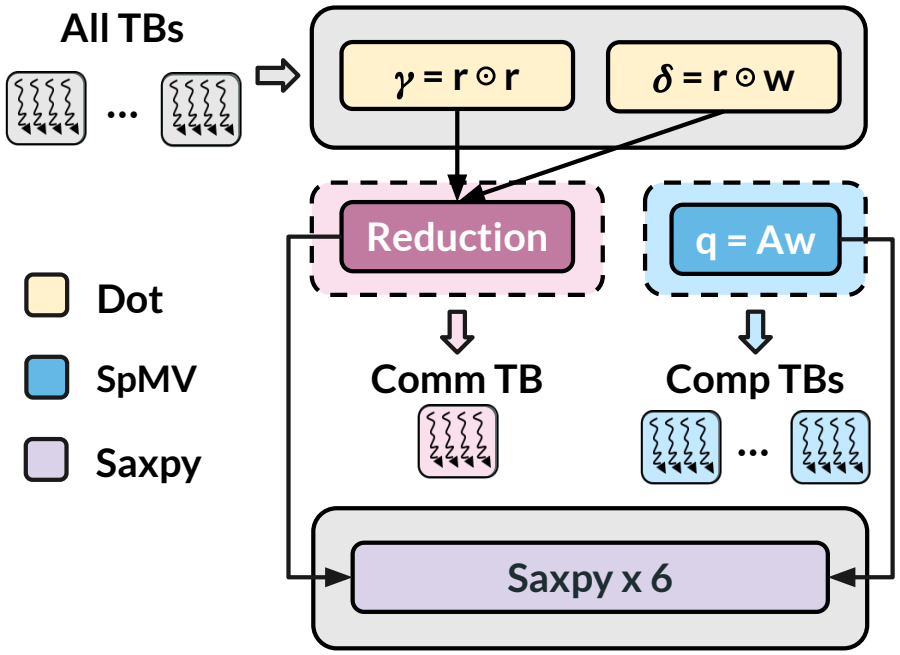


Figure 3.2: Communication/computation overlap with TB specialization for Pipelined CG where reduction of dot products is overlapped with SpMV. Details omitted.

among devices when adapting the method to a multi-GPU setting, Saxpys are vector operations that perform element-wise multiplication and addition on local data and do not require communication. On the other hand, the computation for SpMV needs vector entries on neighboring GPUs, and the dot product requires a global reduction and a follow-up synchronization over all devices to sum the per-GPU dot contributions. While both SpMV and dot products involve communication, prior work has shown that the global reduction and synchronization steps for dot products can become a bottleneck when scaling parallel CG to more nodes [Chronopoulos, 1991, Ghysels and Vanroose, 2014].

The commonly used CG algorithm involves Saxpy(x3), dot product(x2), and SpMV(x1) operations and has few opportunities for communication-computation overlap because of strict dependencies between these operations. To take full advantage of the CPU-free execution model, we also rely on the pipelined CG variant [Ghysels and Vanroose, 2014, Karp et al., 2022], which introduces auxiliary vectors to break up dependencies allowing for the dot product reductions to be overlapped

with the computation for SpMV, at the expense of three additional Saxpys. Furthermore, because the dot products are now also independent of each other, their communication can be implemented as a single reduction, thus, removing a costly global synchronization step [Ghysels and Vanroose, 2014]. We refer to the traditional and pipelined variants as *Standard CG* and *Pipelined CG* respectively, and implement our design for both variants.

Figure 3.2 illustrates the communication/computation overlap we employ in our Pipelined CG implementation, where the global reduction of dot products is overlapped with SpMV. It is important to note that the *communication* TB does not at all times only communicate; it exclusively handles communication *only* when overlap can be achieved. In parts of execution where no overlap is possible, it joins the *compute* TBs to help with the computation.

Listing 3.1 shows the pseudocode of the CPU-free Pipelined CG that uses this overlapping scheme. The traditional Pipelined CG implementation would launch up to nine compute kernels every iteration to perform Saxpy(x6), dot products(x2), and SpMV(x1) operations. Our design leverages persistent kernels not only to fuse those nine operations into a single fat kernel but also to move the time loop to the GPU. In Listing 3.1, step ❶ shows the time loop running on the device. Step ❷ leans on the idea of *TB specialization* for communication-computation overlap by reserving one thread block for the global dot reductions (communication) while the remaining TBs handle SpMV (computation). Step ❸ uses a GPU-initiated reduce call to sum over the local dot contributions. ❹ and ❺ remove the CPU's involvement in synchronization by using grid sync to sync *within* the device and a GPU-side global barrier to sync *across* devices.

```
_
    __global__
     void CPU_Free_PipelinedCG(...) {
    // ❶ Time loop on GPU
    while (iter++ < num_iterations) {

        __local__
            dot(r, r, gamma)
        __local__
            dot(r, w, delta)
        ...
```

```
        // ❷ Specialize one TB for communication
        if (TB_index == 0) {
            // ❸ Multi-GPU reduction
            sum_
                reduce(gamma, delta)
        } else {
            SpMV(A, w, q)
        }
        // ❹ Sync within device
        grid.sync()
        ...
        saxpy(...) //x6
        // ❺ Sync across devices
        multi_gpu.sync()
}}
```

Listing 3.1: Pipelined CG kernel using CPU-free model

3.3.1 Implementation

To implement the CPU-free model on NVIDIA GPUs, we utilize the CUDA Cooperative Groups API and NVSHMEM to enable communication and computation entirely within the kernel. It is worth noting that this approach could also be applied to AMD GPUs since ROCm supports the Cooperative Groups API, and ROC_SHMEM is functionally equivalent to NVSHMEM [Hamidouche and LeBeane, 2020].

CUDA Samples implements a persistent kernel multi-GPU CG solver with Unified Memory [NVIDIA, 2022]. We use this sample as the starting point of our implementation but modify it heavily by opting to use NVSHMEM for communication.

To partition the domain, we equally divide the vectors across the GPUs and allocate the chunks as NVSHMEM symmetric objects. We note that NVSHMEM requires all symmetric object allocations to be the same size. To handle the case when the number of rows is not exactly divisible by the number of GPUs, we allocate a few redundant rows which are ignored in all computational kernels. For the sake of simplicity, we elect to keep a copy of the matrix on each GPU and partition it by

splitting it into logical chunks.

For the CPU-free versions, we launch the persistent kernel using *nvshmemx_collective_launch*, the cooperative launch utility function provided by NVSHMEM. Launching with this routine is required when the kernels use NVSHMEM collective or synchronization APIs, which are extensively employed in our code.

Synchronization

We use the Cooperative Group API's *grid.sync()* function to synchronize all threads within the device. To synchronize across devices, we utilize NVSHMEM's *nvshmem_barrier_all()* function. For the CPU-controlled baselines, we use the host-side stream-based API equivalent *nvshmemx _barrier_all_on_stream()*.

Communication

All communication uses NVSHMEM calls. To get the elements on neighbor GPU's vectors for SpMV, both CPU-controlled and CPU-free versions use the *nvshmem_double_g()* device-initiated call, a blocking call that fetches a single double from a neighbor GPU. For reducing across the dot products, the CPU-free versions use the *nvshmem_sum_reduce_block()*. We also experimented with the warp and thread-level reduction variants but found them to perform worse than the block-level version. The CPU-controlled baselines use the host-side stream-based equivalent *nvshmemx_sum_reduce_on_stream()*. We use block-level APIs for CPU-free versions because that is an explicit advantage of using GPU-side NVSHMEM calls, as block-level APIs have no host-side equivalents. We also note that using NVSHMEM collective APIs on the device requires a *cooperative kernel launch*, making it, in essence, a persistent kernel.

Limitations

The CPU-free execution model suffers from occupancy limitations set by the Cooperative Groups API due to its persistent nature. Device-wide barriers are only supported in cooperative launches that do not oversubscribe the number of thread

blocks, meaning it is impossible to request more blocks than the device can run concurrently. For larger domains, such distribution of work is instead delegated to the programmer (or software), who is required to partition the domain into successive tiles and iterate over them. This limitation is imposed to guarantee the co-residency of the thread blocks, as synchronizing across the grid would otherwise be impossible.

3.4 Evaluation

Table 3.1: SuiteSparse matrices used in CG evaluation

Matrix	Rows	NNZ	NNZ/Rows	Speedup
Queen_4147	4,147,110	329,499,284	79.45	3.33x
Bump_2911	2,911,419	127,729,899	43.87	2.15x
Flan_1565	1,564,794	117,406,044	75.03	2.59x
audikw_1	943,695	77,651,847	82.28	0.94x
Serena	1,391,349	64,531,701	46.38	1.4x
Geo_1438	1,437,960	63,156,690	43.92	1.91x
Hook_1498	1,498,023	60,917,445	40.67	1.62x
ldoor	952,203	46,522,475	48.86	1.04x
StocF-1465	1,465,137	21,005,389	14.34	1.3x
crankseg_2	63,838	14,148,858	221.64	1.0x
hood	220,542	10,768,436	48.83	1.34x
bmwcra_1	148,770	10,644,002	71.55	1.15x
crankseg_1	52,804	10,614,210	201.01	1.0x
G3_circuit	1,585,478	7,660,826	4.83	2.23x
consph	83,334	6,010,480	72.13	1.3x
tmt_sym	726,713	5,080,961	6.99	2.8x
ecology2	999,999	4,995,991	5.00	2.56x
thermomech_dM	204,316	1,423,116	6.97	1.94x
CPU-Free Pipelined Speedup over CPU-Controlled (geo. mean)				1.63x

This section presents the performance scaling of CPU-free execution over CPU-controlled execution for Standard and Pipelined variants of the Conjugate Gradient

solve. The experiments presented here were conducted on NVIDIA HGX machines with 8 NVIDIA A100s connected through NVLink with CUDA toolkit version 11.8 and driver version 495.29.05. The NVSHMEM library version is 2.7.0 with OpenMPI 4.1.4. We repeat each experiment 5 times and report the minimum.

From the SuiteSparse Matrix Collection, we select several symmetric positive definite (SPD) matrices that can converge in a CG solver [12]. We select a wide range of matrices to showcase as broad application scaling behavior as possible. The matrices are summarized in Table 3.1, sorted by the number of non-zeros.

We compare our CPU-free versions against CPU-controlled baselines as explained below:

- **CPU-Controlled Standard CG**: Standard CG with discrete kernels and CPU-initiated communication.

- **CPU-Controlled Pipelined CG**: Pipelined CG with discrete kernels and CPU-initiated communication. Uses concurrent GPU streams for overlap. One stream computes SpMV while the communication stream performs the dot reductions. The communication stream is launched with a higher priority so that it is always scheduled first.

- **CPU-Free Standard CG**: Standard CG with the CPU-free model.

- **CPU-Free Pipelined CG**: Pipelined CG with the CPU-free model. Uses TB specialization for overlap as described in Section 3.3.

Additionally, we implement **Single GPU Standard**, a pure single GPU version of Standard CG with no communication. We use this reference version for reporting speedup. We note that we treat the CPU-controlled baselines fairly to the best of our ability. All versions share the same control flow, placement of within-device and across-device barriers, computational routines with a thread block size of 1024, and communication calls. The only difference between the versions is whether they use the host or device-side APIs.

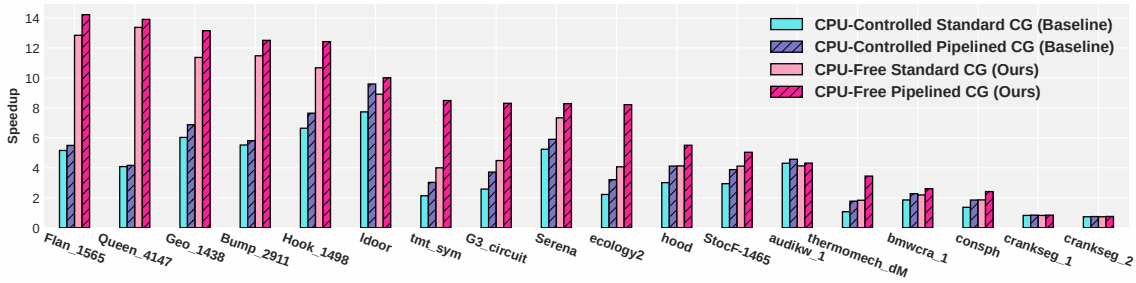


Figure 3.3: Speedup over Single GPU Standard CG of various sparse matrices on 8 NVIDIA A100 GPUs

While, in practice, the CG algorithm is often coupled with a preconditioner for faster convergence [Anzt et al., 2017], we do not apply it as it is orthogonal to our focus on communication. Additionally, we run both *Standard* and *Pipelined* CG algorithms for a fixed number of iterations (5000) and not necessarily to convergence. Even though these algorithms have different convergence properties, we consider the numerical stability of either method to be outside the scope of this work.

3.4.1 Scaling Study

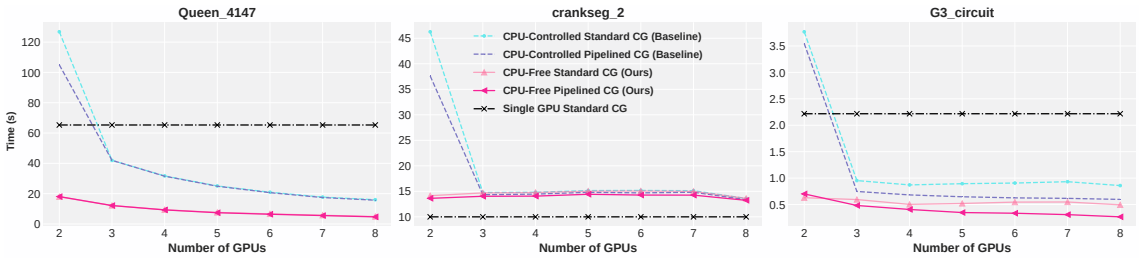


Figure 3.4: Strong scaling study of CG for selected matrices with large, medium, and small sizes

Figure 3.3 shows the speedups achieved by all versions over the *Single GPU Standard* variant. Regardless of the CG variant, CPU-free versions outperform CPU-controlled baselines on almost all matrices, but the speedup varies across matrices. As also shown in Table 3.1, CPU-Free Pipelined CG achieves 1.63x geometric mean speedup over CPU-Controlled Pipelined CG while CPU-Free Standard CG gets 1.54x geometric mean speedup over CPU-controlled Standard CG.

For most of the largest matrices (i.e. `Queen_4147`, `Bump_291`) we observe the significant speedup of our versions over the CPU-controlled ones. For these matrices, SpMV takes up the most time, and there is not enough communication to achieve good overlap. We can observe this by looking at `Queen_4147` in Figure 3.4. For both CPU-free and CPU-controlled versions, there is little difference between the Pipelined and Standard variants meaning there is almost no overlap. Another implication of this is that the speedups we observe are a direct consequence of moving the control path to GPU.

For other matrices (`audikw_1`, `crankseg_2`, `crankseg_1`, `ldoor`, `bmwcra_1`), we observed comparatively subpar speedup compared to the larger matrices. While these matrices have a smaller number of non-zeros, we observe that they are irregular. When these matrices are split among GPUs, there is a significant load imbalance; some GPUs spend more time in SpMV than others. The implication is that all versions are constrained by single-GPU performance. We predict that these matrices could benefit from matrix reordering.

On the other end, we observe significant speedups for smaller matrices (`G3_circuit`, `tmt_sym`, `ecology2`). We note that these matrices have the lowest sparsity (nnz/row); thus, communication takes up a more significant portion of the runtime. In this case, good communication-computation overlap can be achieved, which we observe by comparing the speedups for Pipelined and Standard variants.

3.4.2 Comparisons against PETSc

Furthermore, we compare the CPU-Free model against the state of the art PETSc numerical computing library. The PETSc baselines are based on version 3.17.4 of the library. Communication uses CUDA-Aware OpenMPI (v4.1.5) while computation uses NVIDIA's cuSPARSE library.

Figure 3.5 shows the speedups achieved by CPU-Free and PETSc versions over the *Single GPU Standard* variant. Furthermore, Figure 3.6 show the strong scaling behavior on all 18 SuiteSparse matrices.

There are several insights we can glean by looking at the performance data.

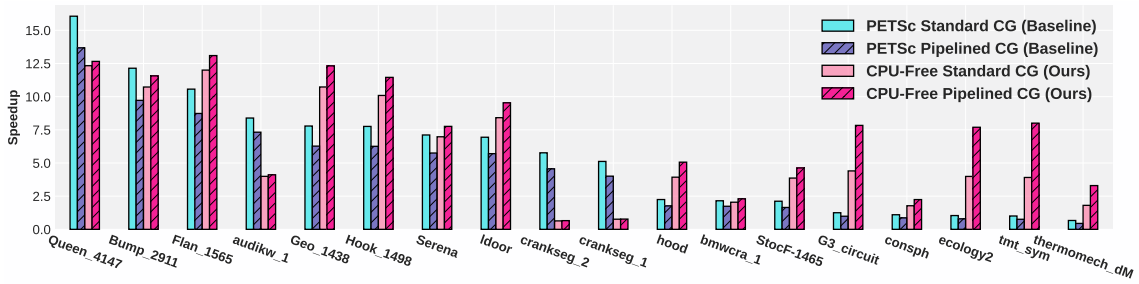


Figure 3.5: CPU-Free and PETSc speedup over Single GPU Standard CG on 8 NVIDIA A100 GPUs

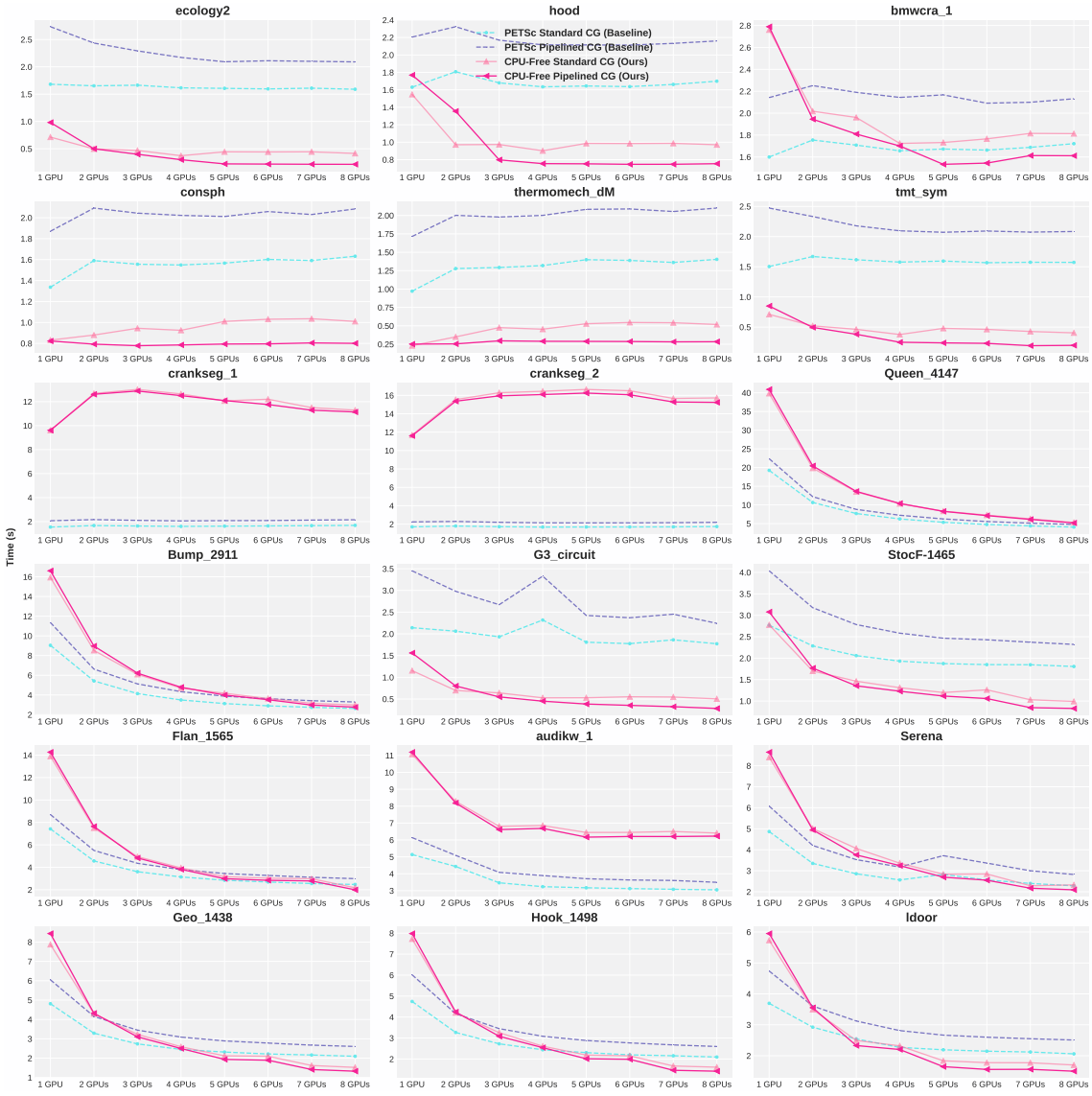


Figure 3.6: Strong scaling comparison between CPU-Free and PETSc on all matrices

First, for several smaller matrices, CPU-Free execution outperforms PETSc even on a single GPU. These matrices are not compute-bound, and by eliminating latencies, CPU-Free versions perform the best. Second, the efficacy of CPU-Free execution for strong scaling can be seen by looking at the scaling behavior for several larger matrices. For these matrices, CPU-Free is constrained by computation and underperforms PETSc at smaller GPU counts. However, as the number of GPUs increases, CPU-Free eventually pulls ahead. Third, CPU-Free can overlap communication with computation, as evidenced by the superior performance of CPU-Free Pipelined CG over CPU-Free Standard CG for several matrices. On the other hand, on all tested matrices PETSc Standard CG consistently underperformed PETSc Pipelined CG. The PETSc FAQ mentions that the MPI implementation should support asynchronous progress threads to effectively overlap the reductions with computation [PETSc, 2023]. To the best of our knowledge, progress threads are not supported in OpenMPI, which could explain the lack of overlap. Experiments with other MPI implementations could help determine how much overlap PETSc can achieve. Finally, there are several compute-bound matrices where CPU-Free underperforms PETSc by a large margin. This is explained by the naive partitioning CPU-Free uses whereby the matrix is partitioned among GPUs by rows and not by the number of non-zeros. This leads to load imbalance, with most of the computation levied upon a single GPU. Reordering the matrix could help in these scenarios. Additionally, we note that CPU-Free computation kernels are unoptimized, given our focus on communication. A notable optimization afforded by persistent kernels is the ability to cache data in shared memory across iterations. We will explore matrix reordering, and across-iteration shared memory caching to optimize computation in future work.

3.5 Related Work

Previous work has shown the benefits of reducing CPU involvement in GPU execution, both along the directions of persistent kernels and GPU-initiated communication. The work by Kshitij et al. [Gupta et al., 2012] was the first to summarize

the persistent kernel concept and discuss how persistent kernels help better load balancing and kernel fusion. Uberkernels that result from the fusion of multiple stages of a dynamic application are proposed in WhippleTree, mostly focusing on graphics workloads [Steinberger et al., 2014]. More recently, work by Zhang et al. [Zhang et al., 2023] demonstrates significant speedup over state-of-the-art single GPU stencil and CG baselines by using shared memory and registers to avoid global memory accesses. Chu et al. [Chu et al., 2019] apply persistent kernels to GPU-based key-value stores to reduce host-induced kernel launch and memory copy overheads leading to better scaling and communication-computation overlap.

Several works have provided the building blocks toward the ultimate goal of freeing the GPU by reducing the involvement of the CPU in the control path. Early works experimented with running the entire network stack on the device but, at the time, suffered from subpar performance and correctness issues due to GPU-NIC interaction [Daoud et al., 2016, Oden et al., 2014a]. Other works added support for GPU networking through CPU helper threads [Gysi et al., 2016, Silberstein et al., 2016]. Agostini et al. [Agostini et al., 2017] introduced GPUDirect Async, which optimized the control path between GPU and NIC by allowing the GPU to trigger and sync CPU-enqueued network transfers. Later, [LeBeane et al., 2017] proposed a NIC hardware mechanism that allowed GPUs to trigger CPU-registered network operations on the NIC from within the kernel, bypassing the CPU, and showed promising results on a simulation. GIO examined the GPU's relaxed memory model for AMD GPUs and fixed the correctness issues stemming from GPU-NIC interaction [Hamidouche and LeBeane, 2020]. Work on NVSHMEM has also optimized for both the data and control paths by providing APIs for fine-grained GPU-to-GPU data movement from within CUDA kernels [Potluri et al., 2015, Potluri et al., 2017, Potluri et al., 2018, Hsu et al., 2020]. While previously NVSHMEM launched CPU proxy threads when communicating over InfiniBand, as of NVSHMEM 2.6.0, the CPU proxy thread can be bypassed, allowing kernel-initiated communication to be issued directly to the NIC [NVIDIA, 2023k].

Other works have used both persistent kernels and GPU-controlled communication to introduce execution models and runtime systems for different applications.

Work by Belviranli et al. [Belviranli et al., 2018] uses a persistent kernel to implement a task-based execution model that treats thread blocks as standalone execution units - a more general form of our block specialization scheme. Chen et al. [Chen et al., 2023, Chen et al., 2022b] present a task-scheduling framework for irregular applications using both persistent and discrete kernels.

These and other GPU-centric works will be discussed further in the next chapter.

3.6 Summary

In this chapter, we propose a fully federated multi-GPU execution model and show its viability on the widely used Conjugate Gradient solver using persistent kernels, thread block specialization, device-initiated communication, and synchronization. By eliminating costly host-side communication and synchronization routines and moving the control path to devices completely, we can achieve significantly reduced communication latencies, allowing better overlap when execution time is bound by data movement across devices. Our experiments on 8 NVIDIA A100 GPUs showed that the CPU-Free model outperforms a CPU-controlled, achieving 1.63x and 1.54x geometric mean speedup over the CPU-controlled baseline for Pipelined and Standard CG solvers, respectively. Additionally, experiments against the state-of-the-art PETSc numerical computing library show better strong scaling behavior and superior communication-computation overlap for most matrices.

This chapter, in part, is a reprint of the material as it appears in the proceedings of the International Conference on Supercomputing 2023 with the title "Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge" by Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. The thesis author was the primary investigator and author of this paper.

Chapter 4

THE LANDSCAPE OF GPU-CENTRIC COMMUNICATION

In the last decade, in response to deficiencies of the traditional model of multi-GPU execution, several advancements, broadly referred to as *GPU-centric* communication, have been proposed. At a high level, these advancements seek to reduce the CPU's involvement in the critical path of execution, give the GPU more autonomy in initiating and synchronizing communication and fix the semantic mismatch between multi-GPU computation and communication. These solutions span a broad spectrum of approaches, including hardware innovations like proprietary GPU-to-GPU interconnects and software mechanisms like GPU-aware MPI. In this chapter, we conduct a comprehensive survey of *GPU-centric* communication. We trace the history of its development, summarize the work done on *GPU-centric* communication methods and discuss their most salient features, including associated benefits and challenges. We organize this chapter as follows:

- In Section 4.1, we begin discussing *GPU-centric* communication. We introduce the terminology and provide a definition for *GPU-centric* communication.

- In Section 4.2, we start discussing vendor mechanisms that provide the means to reduce CPU involvement in the critical path of execution. These mechanisms are used as building blocks for higher-level GPU-centric paradigms, which we discuss in Section 4.3. We list the main communication mechanisms for both *intra-* and *inter-node* setups, discuss their benefits and challenges and rely on existing benchmarking papers to provide insights into their performance.

- In Section 4.3, we identify and discuss the main research paradigms underlying

GPU-centric communication. This section focuses on user-level implementations that leverage the vendor-provided mechanisms discussed in Section 4.2. Among other things, we talk about GPU-aware MPI, GPU-centric collective communication, and CPU-free networking.

- In Section 4.4, we provide an outlook on the field and discuss potential future research directions.

4.1 GPU-Centric Communication

We can loosely define *GPU-centric communication* as *mechanisms for multi-GPU execution which reduce CPU involvement in the critical path*. This is a very broad definition covering a wide spectrum of solutions. These reductions in CPU involvement take the form of optimizations to either the *data* or *control* paths. The data path is the physical channel by which data is relayed, and the control path is where this data is prepared, initiated, and synchronized. Based on this, we categorize GPU-centric communication methods as follows:

- Depending on whether the programmer *initiates* communication from inside a GPU kernel or on the host CPU, we differentiate between *device-initiated* and *host-initiated* methods.

- Depending on where the communication is actually *triggered*, we differentiate between *device-triggered* and *host-triggered* methods. This delineation generally corresponds to the location of the *data path*. A device-initiated method can be triggered by the host under the hood. For example, dCUDA provides MPI-like APIs that can be called from the device but uses CPU threads and host-side MPI to communicate. That would make it a device-initiated but host-triggered method.

- Whether the overall control of multi-GPU execution resides on the GPU or the CPU delineates *device-controlled* and *host-controlled* communication methods.

This corresponds to the location of the *control path*. It is possible for device-initiated and device-triggered methods to be host-controlled. For example, GPUDirect RDMA-based solutions implicitly require CPU synchronization for GPU-NIC memory consistency.

The term *GPU-centric communication* refers to both the vendor-level improvements that grant GPUs autonomy in communication and user-level implementations that leverage those improvements. Since conflating the two could cause some confusion to the reader, we elect to discuss them in separate sections. Section 4.2 focuses on vendor-provided mechanisms, and Section 4.3 discusses how those mechanisms give form to higher-level paradigms.

We also point out the distinction between intra- and inter-node multi-GPU execution. A single GPU-accelerated node comprises a single CPU with multiple GPU cards attached. In a typical scenario, any given GPU is controlled by a single thread, with all threads sharing the same memory and address space. A multi-node system has multiple such nodes where a different process controls each GPU, and memory is not shared between processes. We clarify the distinction between single- and multi-node, as the communication landscape changes depending on the setup used, since inter-node communication requires handling the GPU-NIC interaction and across-process communication. Methods that work for intra-node communication typically do not work for inter-node communication. On the other hand, intra-node methods can handle inter-node communication but may exhibit different performance characteristics and require additional optimizations.

We also note that some methods rely on the proprietary ecosystems of *NVIDIA* or *AMD*. We try to stay as vendor-agnostic as possible, and, for the most part, the ecosystems are similar and provide analogous solutions. However, since some features are only offered by a single vendor, we make note of such vendor-specific approaches.

Finally, there is an unfortunate lack of consistency in existing literature regarding multi-GPU communication methods terminology. For example, several works have used the same term, which can simultaneously refer to multiple communica-

tion methods (e.g., *zero-copy memory*). We hope that providing a comprehensive discussion will help codify the terminology and prevent future ambiguities.

4.2 Vendor Mechanisms

In this section we discuss the vendor-provided mechanisms which allow to reduce CPU involvement in the critical path of execution. These mechanisms are provided by GPU programming model runtimes or as part of the extended APIs. Namely, the communication primitives presented here form the backbone of the higher-level GPU-centric paradigms we discuss in Section 4.3. For clarity of presentation, we divide the discussion across the boundaries of intra- and inter-node communication.

4.2.1 Intra-Node Mechanisms

In this section, we focus on four intra-node GPU-centric communication methods:

- P2P DMA Copies
- P2P Direct Load Stores
- Page-Locked / Pinned Memory
- Unified Virtual Memory (UVM)

Prior to discussing these methods, we introduce the auxiliary technologies which made these methods viable. Namely, we talk about Unified Virtual Addressing (UVA), GPUDirect P2P, and modern GPU-to-GPU interconnects.

Unified Virtual Addressing (UVA)

UVA is a memory management technique introduced in CUDA 4.0 which allows all GPUs within a node and the CPU to share the same unified virtual address space [NVIDIA, 2023a, NVIDIA, 2011]. Prior to UVA, host $\leftrightarrow$ device and device $\leftrightarrow$ device copies had to explicitly specify the direction of transfer. With UVA, the physical memory location can be inferred from pointer values, thus, reducing the overhead of

managing separate memory spaces and enabling libraries to simplify their interfaces [Schroeder, 2011].

GPUDirect 2.0 (Peer-to-Peer)

Along with the introduction of UVA, the CUDA 4.0 release added support for direct Peer-to-Peer communication among GPUs in a single node as long the GPUs shared the same PCIe root complex [NVIDIA, 2011]. Instead of staging data through the host, GPUs could now directly access each other's memory over PCIe, establishing, for the first time, a direct GPU-to-GPU data path. These changes led to two new communication mechanisms: *P2P DMA Copies* whereby a *cudaMemcpy* call would trigger a DMA transfer directly between source and target GPU memories and *P2P Direct Load / Stores* whereby the GPUs could directly access data by dereferencing pointers to the remote GPU buffers. GPUDirect P2P also added support for NVLink (Section 4.2.1) when the latter technology was introduced [NVIDIA, 2023g, NVIDIA, 2012, Rossetti et al., 2016].

GPUDirect P2P provided two main benefits. It eliminated redundant GPU $\leftrightarrow$ CPU copies and host buffers, which were required when the transfers were staged through the CPU. Also, by eliding the need to maintain communication buffers on the host and providing a new communication mechanism (P2P Direct Load / Stores), GPUDirect P2P increased the convenience of multi-GPU programming [NVIDIA, 2012].

We note that P2P DMA Copies can also work without UVA support. If UVA is not enabled, P2P DMA Copies can be performed using the *cudaMemcpyPeer()* variants by explicitly specifying the target GPU. However, P2P Direct Load / Stores will not work without UVA as directly accessing a remote GPU's pointer presumes a unified address space [NVIDIA, 2023a].

Modern GPU-centric Interconnects

NVLink is a proprietary interconnect technology that facilitates high bandwidth and low latency direct access between NVIDIA GPUs. Its design addresses the band-

width limitations of PCIe, which has been observed to be a transfer bottleneck in GPU-accelerated applications [Li et al., 2020, NVIDIA, 2017b]. The first generation of NVLink was introduced along with the Pascal architecture. The P100 GPU had slots for 4 NVLinks, each providing 40 GB/s bidirectional bandwidth for a total of 160 GB/s [Foley and Danskin, 2017]. The next generation, NVLink 2.0, was introduced along with the Volta-based V100 GPU. The V100 increased the number of NVLink slots to 6 and the bidirectional bandwidth of each NVLink to 50 GB/s for a total bidirectional bandwidth of 300 GB/s. The subsequent two generations of NVLink kept the bandwidth of each link at 50 GB/s but increased the number of supported NVLinks per GPU. Ampere and Hopper microarchitecture-based GPUs now provide 12 and 18 NVLink slots totaling 600 GB/s and 900 GB/s bidirectional bandwidth, respectively [NVIDIA, 2023i]. Additionally, NVLink can also connect GPUs with the CPU but this feature is only implemented in IBM Power8 and Power9 CPUs [Li et al., 2020].

While direct GPU-to-GPU P2P communication had already been established over PCIe with GPUDirect 2.0, it was heavily bottlenecked by the low bandwidths of PCIe. The introduction of NVLink optimized the bandwidth between NVIDIA GPUs, turning P2P communication into a viable mechanism for intra-node communication and shifting the *data path* heavily in favor of GPUs. A disadvantage of NVLink is that it is not self-routed meaning that if any two given GPUs do not have a direct NVLink connection communication will have to be routed through an intermediate GPU [Li et al., 2020]. This limitation is overcome by NVSwitch [NVIDIA, 2023i], a backboard technology that can implement all-to-all connections between all GPUs. As an example, a DGX-2 node consists of 16 V100 GPUs that are all-to-all connected through NVLink and NVSwitch [NVIDIA, 2023c].

P2P DMA Copies

Prior to the introduction of GPUDirect P2P, communication between any given 2 GPUs on a single node involved a *cudaMemcpy* call from GPU0 to the CPU and then another *cudaMemcpy* from the CPU to GPU1. With the establishment of a

direct GPU-to-GPU data path, programmers could now use a single *cudaMemcpy* call specifying GPU0 and GPU1 source and destination buffers, and the data would flow directly between GPU memories without involving the host CPU. Under the hood, these copies use the GPU's DMA / Copy Engines to perform transfers. It is important to note that DMA copies will bypass the CPU if there is a direct data path through PCIe or NVLink between the communicating GPUs. If there is no such data path or if peer access is disabled, the data copies will be staged through the host.

The *cudaMemcpyAsync* API variant accepts a GPU stream parameter which can be used to specify the GPU stream on which the copy will execute. This can be used to implement communication-computation overlap whereby the copy is launched in a stream dedicated to communication while the computation is launched in a separate concurrently running stream. GPU events in combination with the *cudaStreamWaitEvent* API call can be used to both synchronize the communication and computation streams within a single GPU and synchronize multiple GPUs with each other. Additionally, to make sure that communication buffers are not overwritten, double-buffering techniques are typically used [NVIDIA, 2023a, Sourouri et al., 2014, Harris, 2012b, Kraus, 2021].

P2P Direct Load / Stores

With the introduction of GPUDirect P2P, it became possible for a given GPU to dereference a pointer to memory residing on a remote GPU. This marked a significant shift in the paradigm of multi-GPU execution, enabling direct communication between GPUs using load and store operations from within the kernel. Direct Load / Stores leverage the high levels of parallelism offered by GPUs by allowing each thread to perform operations on remote memory as if it were local. UVA must be enabled for P2P Direct Load / Stores as directly accessing a remote GPU's pointer presumes a unified address space.

There are typically two approaches when implementing communication with Direct Load / Stores. One is to interleave communication with computation within the

same kernel. The other is to implement communication in a separate kernel. The latter strategy is adopted by the leading collective communication libraries NCCL and RCCL (Section 4.3.2). In this case, communication overlap can be achieved by launching the communication kernel in a separate stream.

Direct Load/Store-based communication offers several benefits. First, it allows the programmer to inline communication with computation, potentially reducing code complexity and improving programmer productivity. The programmer no longer has to rely on separate models for communication and computation and can instead combine them within the GPU kernel [Potluri et al., 2015, Potluri et al., 2018, Langer and Dinan, 2021]. Second, Direct Load/Stores utilize the high levels of parallelism offered by the GPU and can achieve higher levels of bandwidth and lower latencies compared to DMA copies [Ben-Nun et al., 2020, Pearson, 2023]. Third, Direct Load/Stores can *implicitly* overlap communication with computation through the GPU's inherent latency hiding capabilities. Given both the high levels of parallelism granted by the GPU and the increasing bandwidth numbers offered by modern interconnects, the GPU has the capability to hide latencies not only to local but remote memory as well [Potluri et al., 2015, Potluri et al., 2018, Potluri et al., 2017]. This is another boon for the programmer as the method of achieving overlap is shifted from a manual software-based approach implemented by the programmer through streams and events to an automatic hardware-based overlap. Since the onus of communication/computation overlap is passed from the programmer to the hardware, another implication is that the overlap will improve as the hardware gets better at hiding memory latencies. Fourth, Direct Load / Stores expand the scope of applications that could be accelerated through multiple GPUs. Traditionally, applications with fine-grained communication patterns achieved poor scalability on multi-GPU systems as computation frequently had to be interrupted and synchronized in order for the CPU to initiate communication. With Direct Load / Stores from within the kernel, GPUs can adapt well to fine-grained communication patterns. Finally, Direct Load / Stores optimize both data and control planes as the communication can both be *initiated* and *transferred* without leaving the GPU. This direction is particularly promising when combined with persistent kernels allowing

GPU execution that is completely autonomous of the CPU. We discuss this line of research in Section 4.3.3.

Despite the improvements conferred by Direct Load / Stores, there are several inherent challenges. First, a fundamental challenge is that communication and computation contend for the same limited resource as they now both require large volumes of GPU threads to make progress. This can be especially problematic when communication is implemented as a separate kernel. If the computation kernel is launched first, it can potentially monopolize all GPU resources preventing the communication kernel from being launched, effectively, eliminating any possibility of overlap. It is possible to alleviate this issue by launching the communication stream with a higher priority so that it is always scheduled first. We note that P2P DMA Copies do not have this issue as they use the GPU's DMA / Copy Engines - a physically separate resource - for communication [Bernaschi et al., 2021]. Second, similar to single-GPU memory accesses, P2P Direct Load / Stores are highly sensitive to memory coalescing with random non-coalesced access performing far worse than coalesced accesses [Ben-Nun et al., 2020]. Such non-coalesced Direct Reads may expose remote memory latencies which are beyond the GPU scheduler's ability to hide, eventually, stalling execution. On a similar note, sporadic non-coalesced Direct Writes at sub-cacheline granularities may dramatically underutilize the interconnect [Muthukrishnan et al., 2021].

Page-Locked / Pinned Memory

By default, memory allocated on the host using *gpuMalloc()* is allocated as *pageable* and is not accessible to the GPU. When a transfer between pageable host memory and device memory is performed, the GPU runtime must first stage the host data through a temporary buffer in *page-locked memory* and then copy the data from page-locked memory to the GPU. To avoid the pageable $\rightarrow$ page-locked memory copy, *gpuMallocHost()* allows allocating page-locked memory directly, skipping the intermediate copy stage. The CPU and all GPUs can then directly access the pointer to this chunk of memory without any extra copies. Because of this, page-locked

memory is also referred to as *zero-copy* memory. Additionally, page-locked memory is also called *pinned* memory [NVIDIA, 2023a, Harris, 2012a].

Several benchmarks have shown that pinned memory can achieve high bandwidth and low latencies, particularly for host $\leftrightarrow$ device transfers [Pearson et al., 2019]. For this reason, it can be used to efficiently coordinate execution between the CPU and GPUs as pinned memory pointers can be accessed directly across the system. Pinned memory has also been used with GPUDirect RDMA to improve inter-node communication [Li et al., 2020]. However, because pinned memory is locked in physical memory, it can consume a significant amount of memory, and excessive allocations can degrade overall system performance [Harris, 2012a].

Unified Virtual Memory (UVM)

Introduced in CUDA 6.0, UVM allows for the allocation of *managed* memory through *cudaMallocManaged()* calls by creating a single address space accessible to all processors within a single node. UVM works by dividing the requested memory into pages that are resident on the CPU. The programmer can access memory on a device without explicit copies. If a memory access is part of a page that is not on the device, the UVM-driven triggers a page-fault that automatically migrates the page to the requesting device. The UVM driver can also evict pages from a given device back to host memory when the total page memory size exceeds device memory [NVIDIA, 2023a].

UVM provides several benefits in regard to programmability. First, programmers are exposed a single unified address space that they can access as if the whole allocated chunk of memory is resident on a single GPU. Any copies occurring around the system are implicit and hidden from the programmer's view. Additionally, UVM allows memory oversubscription whereby more memory can be allocated than all GPU device memory combined. This is possible since most of the memory can stay on the CPU and be paged in whenever a given device requests it [NVIDIA, 2021, Shao et al., 2022].

However, UVM offers generally subpar performance due to a problem known

as *page-thrashing*. As devices make more and more accesses, UVM migrates pages back and forth which end up *thrashing* between GPU and host memories, severely impacting performance. This is especially problematic in applications with irregular access patterns where each new access is likely to bring in a new page and migrate an old one. While optimization techniques like prefetching data to the GPUs to prevent thrashing can improve performance, in general, UVM performs well in specific cases and severely outperforms most mechanisms in others.

4.2.2 Inter-Node Mechanisms

We now discuss GPU-centric communication across nodes. These methods address the two primary challenges in inter-node communication: 1) GPU-NIC interaction and 2) Inter-process communication.

CUDA IPC

Previously, GPUDirect P2P allowed direct P2P communication across GPUs within a single node but was restricted to single process setups. Pointers could not be accessed across process boundaries, so memory copies between GPU buffers had to go through the host, creating a bottleneck. To overcome this limitation, CUDA 4.1 introduced CUDA Inter-Process Communication (IPC), which enables processes on the same machine to access device buffers of other processes without additional copies [NVIDIA, 2017a]. With CUDA IPC, memory handles are created and passed between processes using standard IPC mechanisms, resulting in lower latencies than staging copies through the host. However, the overhead of creating memory handles can be significant and may offset the latency benefits. [Potluri et al., 2012]. An analogous technology called ROCm IPC is offered by AMD.

While CUDA IPC is technically an intra-node communication mechanism, its primary use lies in efficiently adapting inter-node communication mechanisms to intra-node setups. Namely, GPU-Aware MPI implementations, NVSHMEM / ROC_SHMEM, and NCCL / RCCL, all of which create one process per GPU, use IPC under the hood for intra-node communication.

GPUDirect 1.0 (Shared GPU-Sysmem)

Introduced in CUDA 3.1, GPUDirect 1.0 allowed GPUs and NICs to share the same pinned memory region. Prior, the pinned memory regions in system memory for GPUs and the NIC were separate. By implication, to communicate GPU data across nodes, the GPU first copies the data to its pinned memory region, the CPU then copies it to NIC's memory region, and, finally, the NIC sends it across the network. The intermediate CPU-initiated copy from GPU $\rightarrow$ NIC pinned memory regions introduces CPU overhead and increases the latency for GPU communication. GPUDirect 1.0 introduced a shared memory GPU-NIC pinned memory region, thus, avoiding the intermediate CPU-initiated copy. Experimental evaluation of molecular dynamics software across eight single-GPU nodes has demonstrated improvements granted by GPUDirect 1.0 [Rossetti et al., 2016, Shainer et al., 2011].

GPUDirect RDMA

With the introduction of GPUDirect RDMA in CUDA 5.0, direct communication between NVIDIA GPUs across nodes became feasible. GPUDirect RDMA facilitates a direct communication channel between GPUs and third-party devices through standard PCIe features. The technology exposes segments of GPU memory on the PCIe memory resource, referred to as the Base Address Register (BAR) region. This enables Network Interface Cards (NICs) to directly read/write GPU memory without routing through the host [NVIDIA, 2023d]. Analogously, AMD offers ROCm RDMA (previously called ROCnRDMA) [AMD, 2023d, AMD, 2023b].

GPUDirect RDMA provides several optimizations to the data path, namely by eliminating additional copies to host memory, reducing inherent latencies stemming from GPU-NIC interaction, increasing bandwidth and reducing CPU overhead. Support for GPUDirect RDMA has been integrated into several leading communication libraries including *GPU-aware* MPI implementations, NCCL and NVSHMEM.

A significant limitation of GPUDirect RDMA is that there are no guarantees of consistency between GPU and NIC memories while a kernel is running. Consistency is guaranteed only by returning control to the CPU by tearing down the kernel and

launching a new kernel, thus, limiting communication to kernel boundaries. This also implies that combining persistent kernels with GPU-initiated inter-node communication will inevitably lead to data correctness issues [NVIDIA, 2023d]. Chu et al. get around this limitation by issuing a PCIe read from the NIC to GPU memory which flushes the previous NIC writes to the GPU and guarantees memory ordering [Chu et al., 2019]. Since version 11.3, CUDA also offers the *cudaDeviceFlushGPUDirectRDMAWrites()* function which can be used to enforce consistency similarly [NVIDIA, 2023b, Davide Rossetti, 2021]. While useful, CUDA still relies on the CPU to enforce GPU-NIC consistency. AMD, on the other hand, has explicitly corrected the GPU-NIC consistency issues in the context of device-side communication from persistent kernels and integrated the proposed fixes into ROC_SHMEM [Hamidouche and LeBeane, 2020]. We further discuss this issue in the context of CPU-free networking in Section 4.3.3.

Gdrcopy

GPUDirect RDMA permits third-party devices, such as NICs, to access GPU memory directly. The set of APIs used for this functionality is also sufficiently versatile to expose GPU memory to the CPU more broadly. Gdrcopy, an open-source library, provides a means of mapping GPU memory to user-space, thus allowing access to it as if it were regular host memory. GDRCopy also provides optimized copy APIs and is widely used in high-performance communication runtimes [NVIDIA, 2023e].

Gdrcopy provides a low-latency alternative for transferring data between the CPU and GPU with minimal overhead. In contrast, cudaMemcpy uses the GPU DMA / Copy Engines to move data between the CPU and GPU memories, resulting in latency overheads and lower performance for small data sizes. With GDRCopy, the CPU can directly access GPU memory through BAR mappings, facilitating low-latency copies between GPU and CPU memories. [NVIDIA, 2023e, Davide Rossetti, 2021]. AMD does not explicitly provide an alternative technology, however, analogous capabilities are offered by the PCIe LargeBar feature of the ROCm driver [Shafie Khorassani et al., 2021, Khaled Hamidouche, 2018].

Despite the benefits, there are some caveats to using Gdrcopy. While DMA copies rely on the GPU's DMA / Copy Engines, Gdrcopy consumes CPU core cycles and hardware buffers. Furthermore, enabling Gdrcopy requires installing and loading a separate kernel module which may add extra complexity. Additionally, Gdrcopy is optimized for small data sizes and may not perform well for larger transfers [NVIDIA, 2023e, Shafie Khorassani et al., 2021, Davide Rossetti, 2021].

GPUDirect Async

While previous GPUDirect technologies focused on improving the data path, GPUDirect Async optimizes the control path between the GPU and the NIC. Introduced in CUDA 8.0, it enables GPUs to initiate and synchronize network transfers, thereby reducing the CPU's involvement in the critical path. GPUDirect Async works by having the CPU pre-register messages, which the GPU kernel can then trigger by ringing a doorbell on the NIC. As a result, the GPU can continue executing while the communication is being triggered, rather than needing to stop for the CPU to initiate the communication, as was previously necessary [Agostini et al., 2017, Agostini et al., 2018].

Although GPUDirect Async has led to improvements in efforts to move the control path away from the CPU, it still does not completely transfer the control path to the GPU since communication is limited to kernel launch boundaries. Essentially, the GPU can only initiate messages previously registered by the CPU. Further improvements to GPUDirect Async are implemented as part of the IBGDA transport in the NVSHMEM library (Section 4.3.3).

4.3 GPU-Centric Communication Paradigms

We now discuss the main GPU-centric communication paradigms that have sprouted in recent literature.

4.3.1 GPU-Aware MPI

Given that MPI is the de facto *lingua franca* of HPC, much effort has gone into making MPI communication interoperable with GPU programming models. This work has culminated in *GPU-Aware* MPI, typically defined as *MPI implementations that can differentiate between host and device buffers.* Prior to *GPU-Aware MPI*, all multi-GPU communication had to be staged through the host incurring a device → host copy on the source GPU and a host → device copy on the target GPU. Using a *GPU-Aware MPI* implementation, on the other hand, a programmer can supply device buffers as parameters to the MPI call allowing communication to use the direct GPU-to-GPU data path established by GPUDirect RDMA or ROCnRDMA. In the process, *GPU-awareness* eliminates redundant host ↔ device copies and simplifies the communication code by eliding the need for host buffers.

MVAPICH2 was the first MPI implementation to begin actively integrating GPU-awareness into its runtime. Early work done prior to the introduction of GPUDirect RDMA added basic GPU-awareness whereby users could initiate MPI calls with buffers residing on the GPU. The resulting communication still had to be staged through the host, but this was done transparently by the library and optimized using pipelining schemes for the host ↔ device and device ↔ device transfers. These pipelining schemes were made possible by UVA, which allowed the library to differentiate between host and device pointers without relying on user hints. The ensuing GPU-awareness led to performance improvements over the GPU-oblivious version [Wang et al., 2011a, Wang et al., 2011b, Wang et al., 2014]. A follow-up work used CUDA IPC to optimize intra-node transfers, which prior had to be staged through buffers in host memory [Potluri et al., 2012]. Eventually, support was added for GPUDirect RDMA over the rendezvous protocol allowing transfers to bypass the host and eliminate redundant host ↔ device copies. This reduced latencies; however, bandwidth was limited due to existing architectural limitations [Potluri et al., 2013b]. A subsequent work added support for GPUDirect RDMA over the eager protocol rectifying the bandwidth limitation and further reducing latencies. Additionally, a new loopback mechanism and an early version of Gdrcopy were used

to eliminate expensive host ↔ device *cudaMemcpy*s [Shi et al., 2014]. Another work extended point-to-point MPI calls to support GPUDirect Async allowing the GPU to progress the communication enqueued by the CPU, thus, optimizing the control path [Venkatesh et al., 2017]. Other works have also increasingly focused on adding UM-awareness to MVAPICH2-GDR [Banerjee et al., 2016, Hamidouche et al., 2016, Manian et al., 2019].

While MVAPICH2 certainly paved the way for GPU-aware MPI, other leading MPI implementations have also integrated support for GPU-awareness. OpenMPI [OpenMPI, 2023a, Wu et al., 2016], HPE Cray MPICH [HPE, 2021], MVAPICH2 and IBM Spectrum MPI [MPI, 2021] natively support CUDA-awareness. OpenMPI also supports CUDA through UCX. Given the wider deployment of NVIDIA GPUs, the literature on ROCm-aware MPI is sparse. Existing MPI implementations with the sole exception of MVAPICH2 rely on UCX for ROCm-awareness [AMD, 2023a, AMD, 2023c, OpenMPI, 2023b]. On the other hand, Khorassani et al. provide a native ROCm-aware runtime for MVAPICH2 which outperforms OpenMPI with UCX on a cluster of AMD GPUs [Shafie Khorassani et al., 2021]. To the best of our knowledge, Spectrum MPI cannot be made ROCm-aware.

Despite the benefits of GPU-aware MPI, there are inherent challenges with MPI-GPU integration. Most importantly, there is a fundamental *semantic mismatch* between MPI and GPU programming models. As discussed in Section 2, GPUs operate on the concept of streams which are command queues that guarantee ordering among GPU operations. The GPU scheduler ensures that kernels and other operations launched on a stream execute in the order they were enqueued and do so with correct data dependencies. Since kernel launches are asynchronous and do not block the host, GPU runtimes can pipeline kernel launches and overlap the launch latencies behind kernel execution. The *semantic mismatch* between the MPI and GPU models is that MPI has no awareness of GPU streams. As a result, it is not possible to enqueue an MPI call on a given GPU stream or for a GPU stream to wait on the completion of a pending MPI routine. By implication, interlacing MPI calls with GPU kernels will require host-blocking synchronizations in order to maintain data correctness. For example, before initiating an MPI send, the programmer

has to block the host to synchronize all streams which operate on the send buffer. Similarly, waiting on completion of pending MPI communication will also require host-blocking synchronization. In either case, these forced synchronizations impair kernel launch pipelining, prevent opportunities for overlap and force the programmer into alternating bulk phases of communication and computation. [Dryden et al., 2018, Zhang et al., 2021].

The *semantic mismatch* between MPI and the GPU ultimately stems from trying to combine a CPU-centric communication runtime that has no concept of streams with GPU-based computation, which, by design, relies on streams for concurrency, synchronization, and overlap. This fundamental issue is unlikely to be addressed by piecemeal solutions. We see two possible non-mutually exclusive paths for resolving the semantic mismatch. The first is making MPI runtimes stream-aware by adding an explicit stream parameter to MPI routines. This would solve the issue of impaired kernel launch pipelining and allow MPI calls to seamlessly integrate into GPU runtimes. The second is providing the option of device-initiated MPI calls. This would reduce the programmer burden of juggling two distinct programming models and, additionally, provide implicit communication-computation overlap. Both directions have been explored in the literature on a limited scale. The FLAT compiler automatically converts device-side MPI calls to their host-side equivalents [Miyoshi et al., 2012]. dCUDA implements device-side operations with MPI semantics but uses CPU helper threads for the actual communication. They rely on the GPU's inherent memory latency hiding capabilities to *implicitly* overlap communication with computation, ultimately outperforming a GPU-aware MPI baseline [Gysi et al., 2016]. Namashivayam et al. explore new communication schemes to introduce GPU stream-awareness in MPI. They use the *triggered operations* feature on HPE Slingshot 11 interconnect, allowing the CPU to enqueue communication and synchronization operations to the NIC, which the GPU can then trigger. This reduces CPU involvement in the critical path and eliminates expensive synchronizations. While intra-node experiments show some performance benefits, the proposed scheme falters in intra-node setups as progress threads need to be used to emulate deferred execution semantics [Namashivayam et al., 2022]. A

follow-up work eliminates progress threads for intra-node communication, opting to use P2P Direct Load Store-based GPU kernels and GPU IPC-based mechanisms instead. The evaluation shows performance improvements over stream-oblivious MPI baselines [Namashivayam et al., 2023].

4.3.2 GPU-Centric Collectives

As deep learning models get ever larger, their compute requirements necessitate deploying training across multiple GPUs. In line with this, many works use GPU-centric mechanisms to optimize multi-GPU training. Given the prevalence of collective communication in deep learning training, several works attempt to provide efficient collective primitives.

NCCL

NCCL (NVIDIA Collective Communication Library) is a software library that offers topology-aware collective primitives for inter-GPU communication [NVIDIA, 2023f]. Additionally, since version 2.7, NCCL also provides point-to-point communication APIs. AMD offers an analogous library called RCCL.

Traditional *CPU-centric* multi-GPU collectives were implemented using GPU kernels for local reductions and CPU-initiated copies among the GPUs. This approach incurs several kernel launch and communication call latencies and, additionally, requires intermediate buffers on the host. NCCL takes a different approach by implementing the communication and computation for the collective together in a single kernel [NVIDIA, 2016]. The first version of NCCL, NCCL 1.0, only supported intra-node communication, which was implemented using *P2P Direct Load / Stores*. NCCL 2.0 added support for inter-node communication, which relies on GPUDirect RDMA and proxy threads running on the CPU [Jeaugey, 2017, Jeaugey, 2019].

NCCL has gained significant traction in the HPC community, especially in the realm of deep learning frameworks, owing to its efficient collectives and streamlined API. It has been integrated as a communication backend for several state-of-the-art deep learning frameworks, including Pytorch, Tensorflow, MXNet, Caffe, CNTK,

and Horovod [Weingram et al., 2023].

NCCL offers several benefits. First, NCCL has been shown to achieve high levels of bandwidth and parallel efficiency [NVIDIA, 2016, Kraus, 2021, Bernaschi et al., 2021, Weingram et al., 2023]. Second, NCCL is topology-aware, thus, relieving the programmer of optimizing for the topology. Third, NCCL is natively compatible with the CUDA programming model and can seamlessly integrate with it. The NCCL API calls come with a stream parameter, allowing communication-computation overlap by enqueueing the collective communication on a separate stream.

Despite its benefits, NCCL does come with its own set of inherent challenges. First, a fundamental issue of using GPU threads for communication *and* computation is that the two routines now contend for the same limited resource. In this case, if computation is scheduled ahead of the NCCL collective, it can monopolize all GPU resources, effectively serializing the collective behind the computation. One workaround around this is to launch the NCCL collective on a higher-priority stream such that it is always scheduled first [Bernaschi et al., 2021, NVIDIA, 2016]. We also discussed this issue in the context of *P2P Direct Load / Stores* in Section 4.2.1.

Further Work on Collectives

Awan et al. use the at the time newly released NCCL to implement efficient intra-node *MPI_Bcast* as part of MVAPICH2-GDR. Since early versions of NCCL did not support inter-node communication, hierarchical collective mechanisms are proposed to support scaling out across nodes [Awan et al., 2016]. Subsequent work challenges NCCL's hegemony of deep learning collective communication workloads by proposing new pipelined designs for the *MPI_Bcast* collective. The proposed design outperforms NCCL2 broadcast for small message sizes and offers comparable performance for larger ones [Awan et al., 2019]. Blink is a collective communication library with the express goal of achieving optimal link utilization. To do this, Blink detects the underlying topology, models the topology as a graph, and then uses a technique known as *packing spanning trees* to dynamically generate communication

primitives. As a result, Blink was shown to reduce model training time on an image classification task by 40% compared to NCCL2 [Wang et al., 2019]. Dryden et al. present Aluminum, a GPU-aware library for large-scale training of deep neural networks. Namely, Aluminum aims to rectify the inherent drawbacks of both NCCL and MPI. Aluminum extends NCCL with tree-based algorithms to avoid the latency bottlenecks of its default ring implementation. Additionally, it adds support for non-blocking NCCL allreduce operations. For MPI, Aluminum gets around forced synchronizations stemming from the MPI-GPU semantic mismatch by associating a single GPU stream with an MPI communicator and synchronizing with respect only to that stream. The resulting optimizations bring about speedups compared to GPU-aware MPI and NCCL-based implementations [Dryden et al., 2018]. Soyturk et al. present ComScribe, a tool that allows monitoring NCCL collective and P2P communication. However, it is limited to a single node and relies on the deprecated *nvprof* tool [Soyturk et al., 2021]. In a recent work, Chen et al. use CUDA IPC to optimize MPI intra-node all-to-all communication. They use CUDA IPC to pass device buffers across process boundaries and then communicate using *P2P Direct Load / Stores* for small messages and *P2P DMA Copies* for larger ones [Chen et al., 2022a].

4.3.3 CPU-Free Networking

As the trend toward GPU-centric communication and greater GPU autonomy continues to accelerate, several works have suggested migrating most or all of the networking stack to the kernel. This is typically done by launching a single long-running persistent kernel and moving both the data and control paths to the GPU.

In the earliest work, GGAS proposes changes to network devices to implement a unified global address space that allows moving the control path entirely to the GPU. This is accomplished by using a persistent kernel that contains the computation, communication, and synchronization all on the device-side. While the work was the first of its kind and showed performance improvements compared to a CUDA-Aware MPI baseline, the experiments were conducted on two GPU nodes with one GPU

each, while the proposed hardware changes were emulated on an FPGA [Oden and Fröning, 2013]. Follow-up work showed that GGAS, by virtue of eliminating CPU involvement in the control path, can achieve further performance improvements and reduce energy usage compared to CPU-controlled baselines [Oden et al., 2014b, Klenk et al., 2014b].

Several more works on CPU-free networking followed. Oden et al. use GPUDirect RDMA to allow GPUs to directly interface with Infiniband network devices without the involvement of the host CPU. They do this by mapping the entire Infiniband context to the device-side and using the GPU to generate and send work requests to the HCA. However, because of slow single-thread work request generation performance on the GPU, the proposed changes deteriorated performance compared to CPU-controlled baselines [Oden et al., 2014a]. Follow-up work ameliorated these performance limitations and showed much more promising results [Klenk et al., 2014a, Klenk et al., 2015]. Another work combines the proposed GPU-side Infiniband Verbs with CUDA Dynamic Parallelism to optimize the bottleneck of intra-kernel synchronization [Oden et al., 2014c]. GPUrdma also implements Infiniband on the GPU and proposes a GPU-side library for direct communication from within persistent GPU kernels with zero CPU involvement. The proposed design outperforms a CPU-controlled baseline on a series of microbenchmarks but runs into correctness issues resulting from the use of persistent kernels and GPU-NIC interaction [Daoud et al., 2016]. As discussed in Section 4.2.2, these correctness issues stem from the fact that memory consistency between GPU and the NIC is guaranteed only at kernel boundaries, and persistent kernels which never synchronize with the CPU violate this guarantee. Silberstein et al. implement GPUNet, which provides GPU-side socket abstractions and networking primitives. GPUNet allows invoking the communication on the GPU but does not fully migrate the control path to the device; instead, it relies on CPU helper threads to perform the actual communication [Silberstein et al., 2016]. A similar approach is adopted by dCUDA, which provides device-side APIs with MPI semantics but translates them to standard MPI calls performed by CPU helper threads [Gysi et al., 2016]. LeBeane et al. categorized GPU networking methods discussing at length their deficiencies.

In response, they propose GPU-TN, a NIC hardware mechanism that allows the CPU to create and register messages with the NIC and the GPU to trigger them from a running persistent kernel [LeBeane et al., 2017]. Another work, ComP-Net, uses embedded GPU microprocessors to offload helper threads from the CPU to the GPU [LeBeane et al., 2018]. While both GPU-TN and ComP-Net show promising performance, they require hardware changes to the NIC and the GPU and, thus, rely on simulation to obtain results.

A distinct direction in research on CPU-free networking is that of GPU-centric OpenSHMEM runtimes, namely, NVIDIA's NVSHMEM and AMD's ROC_SHMEM libraries. The two libraries are fundamentally similar but have some differences. Given its earlier inception, we first discuss NVSHMEM and, in doing so, introduce concepts fundamental to both libraries. In the ROC_SHMEM section, we discuss its important differences from NVSHMEM.

NVSHMEM

NVSHMEM is NVIDIA's implementation of the OpenSHMEM specification for CUDA devices. NVSHMEM is a Partitioned Global Adress Space (PGAS) library that provides efficient one-sided *put / get* APIs for processes to access remote data objects. NVSHMEM supports point-to-point and collective communication between GPUs both within and across nodes [NVIDIA, 2023j].

NVSHMEM works on the concept of a *symmetric heap*. During NVSHMEM initialization, each process that is mapped to a GPU, referred to as a processing element (PE) reserves a block of GPU memory using *nvshmem_malloc()*. In NVSHMEM, all memory allocations must be performed collectively, meaning that all symmetric memory regions within the heap must have identical sizes and must be allocated at the same time. To access remote memory on a different PE, a given PE requires the offset for the symmetric memory as well as the rank of the remote PE.

In addition, NVSHMEM provides APIs for synchronizing a group of PEs. These APIs comprise signal-wait mechanisms that can serve as a means for point-to-point synchronization and collective synchronization calls that can function as global bar-

riers. This feature is particularly important since there is a general lack of kernel-side global barriers, with the CPU typically performing the role of global synchronizer for devices. The capacity for a device to synchronize efficiently across the device without terminating kernel execution is a crucial prerequisite for transferring the control plane to the GPU.

A notable attribute of NVSHMEM is that it offers both host-initiated and device-initiated APIs. The host-initiated APIs expose an optional stream argument that can be used to implement communication-computation overlap. For certain calls, the GPU-initiated variants provide the calls in three granularities: thread, thread block, and warp. The thread variant means that the call should be performed by a single device thread and will be executed by that thread. The thread-block and warp variants require all threads in the corresponding hierarchy to execute the communication call cooperatively. These variants should be called by all threads in the corresponding thread block or warp. A previous performance comparison between the host and GPU-side APIs found negligible differences in performance between the two, with host-side APIs slightly outperforming the device-side variants [Groves et al., 2020]. This study was conducted using an early version of NVSHMEM (0.3.0), which has since seen improvements in GPU-side API performance.

As of version 2.7.0, NVSHMEM introduced the Infiniband GPUDirect Async (IBGDA) transport built on top of GPUDirect Async [NVIDIA, 2023l]. The IBGDA transport allows GPUs to issue inter-node communication directly to the NIC, bypassing the CPU entirely. Without IBGDA, device-side inter-node communication calls are performed through a proxy thread on the CPU that triggers the corresponding NIC operations. This proxy thread consumes CPU resources and creates a bottleneck in achieving peak NIC throughput for fine-grained transfers [Pak Markthub and Howell, 2022]. NVSHMEM with IBGDA support, combined with persistent kernels, enables the complete transfer of both data and control paths to the GPU and marks a significant shift towards fully autonomous multi-GPU execution. However, as discussed in Section 4.2.2, GPUDirect RDMA *only enforces GPU-NIC memory consistency across kernel boundaries.* By implication, a long-running persistent kernel that never synchronizes with the CPU and communicates across nodes will

inevitably run into correctness issues. We see this inherent reliance on the CPU for memory consistency as a significant obstacle toward truly autonomous multi-GPU execution. One workaround is using a callback mechanism whereby the persistent kernel signals the CPU to perform a consistency-enforcing API call (i.e., *cudaDeviceFlushGPUDirectRDMAWrites()*). The efficacy of this solution is unclear and warrants further investigation. Enforcing GPU-NIC memory from *inside the kernel* is supported by ROC_SHMEM, which we discuss next.

ROC_SHMEM

ROC_SHMEM is AMD's implementation of the OpenSHMEM specification for AMD GPUs. ROC_SHMEM offers two communication backends. The first, known as GPU-IB, implements Infiniband on the GPU, similar to NVSHMEM's IBGDA transport. The second, called Reverse Offload (RO), uses host-side proxy threads and offloads communication to the CPU. GPU-IB is the default backend and offers the best performance [AMD, 2023e].

ROC_SHMEM works almost identically to NVSHMEM and offers analogous APIs. However, there are several significant differences.

First, as mentioned in the previous section, NVSHMEM runs into GPU-NIC memory consistency problems when intra-node communication is issued from persistent kernels. ROC_SHMEM, on the other hand, explicitly addresses this issue and guarantees correctness when persistent kernels are being used. Hamidouche et al. analyze the GPU-NIC memory mismatch stemming from the GPU's relaxed memory model and propose changes integrated into ROC_SHMEM [Hamidouche and LeBeane, 2020]. This means that *ROC_SHMEM is the only completely CPU-free communication mechanism that can correctly move the entire flow of multi-GPU execution to the device.*

Second, ROC_SHMEM uses GPU *shared memory* (*local data store (LDS)* in AMD parlance) to store network state for faster access. To the best of our knowledge, this optimization is not implemented in NVSHMEM. While this is most likely beneficial for execution time, the increased shared memory message could limit oc-

cupancy and negatively impact performance [Punniyamurthy et al., 2023, AMD,].

Third, prior versions of ROC_SHMEM required allocating symmetric buffers as uncacheable in order to prevent stale data from being communicated. However, as AMD recently introduced intra-kernel cache flush instructions, the data can be flushed before initiating the network transaction, allowing the data to be cached. No such instructions are provided by NVIDIA, meaning that NVSHMEM buffers are likely allocated as uncacheable [Punniyamurthy et al., 2023].

These differences lead us to the following conclusion: *ROC_SHMEM is explicitly designed with CPU-free execution in mind and implements more optimizations for that setting.* NVSHMEM has also made strides toward CPU-free execution with its IBGDA transport but still relies on the CPU for something as fundamental as GPU-NIC memory consistency. While we have qualitatively compared the two, a quantitative comparison to determine which is better suited for CPU-free networking would be helpful and is the subject of future work.

Applications

In recent years, NVSHMEM has increasingly been integrated as a communication backend into multiple runtimes. PETSc implemented PetscSF, a scalable communication layer based on NVSHMEM, to complement their MPI-based approach, which did not work well with CUDA stream semantics and prevented kernel launch pipelining [Zhang et al., 2021]. Kokkos Remote Spaces, which adds distributed memory support to the Kokkos programming model, uses NVSHMEM as one of its communication backends [Ciesko, 2023, Trott, 2018]. An NVSHMEM implementation of the Kokkos Conjugate Gradient Solver has been shown to outperform the CUDA-aware MPI implementation while also significantly reducing the size of the code base [Maruyama et al., 2020]. Choi et al. use persistent kernels and NVSHMEM to implement a fully GPU-resident runtime system called CharminG that takes inspiration from Charm++ [Choi et al., 2021b]. Livermore Big Artificial Neural Network (LBANN) implements a spatial-parallel convolution using NVSHMEM that outper-

forms MPI and Aluminium implementations [Maruyama et al., 2020]. QUDA, a library for performing computations in lattice QCD, has used NVSHMEM and persistent kernels for improved strong scaling of the Dirac operators [lattice, 2023, Wagner, 2020].

NVSHMEM has also been used in other contexts outside of runtime-based approaches to achieve performance improvements. Chu et al. combine NVSHMEM with persistent kernels to implement a state-of-the-art GPU-based key-value store [Chu et al., 2019]. Xie at al. use NVSHMEM to implement a single-node multi-GPU sparse triangular solver (SpTRSV) solver which achieves good performance scalability compared to a UVM-based design [XIE et al., 2021]. Ding et al. combine persistent kernels with NVSHMEM to achieve impressive performance for a sparse triangular solver (SpTRSV) on single- and multi-node setups. Atos implements both persistent and discrete kernels with NVSHMEM-based communication to achieve state-of-the-art performance on multi-GPU BFS both within and across nodes [Chen et al., 2022b]. Wang et al. propose MGG, a system design that accelerates Graph Neural Networks (GNNs) on multi-GPU systems using a GPU-centric software communication-computation pipeline that uses NVSHMEM for fine-grained communication [Wang et al., 2023]. Ismayilov et al. use persistent kernels and device-side NVSHMEM to implement fully GPU-side Jacobi 2D/3D and CG solvers which outperform CPU-controlled baselines. They also explicitly reserve some thread blocks for communication while the remaining concurrently running ones handle computation. This technique which they call *TB specialization*, is used to achieve *explicit* device-side communication-computation overlap [Ismayilov et al., 2023]. Punniyamurthy et al. use ROC_SHMEM and persistent kernels to overlap embedding operations with collective communication in deep learning recommendation models [Punniyamurthy et al., 2023].

Discussion

There are several reasons why we believe CPU-free networking, and GPU-centric OpenSHMEM in particular, shows promise. First, it addresses the issue of CPU-

induced latency barriers that are caused by kernel launch and memory copy overheads. These barriers become more significant as the number of GPUs increases and computation per GPU decreases. In latency-bound settings, traditional CPU-controlled implementations are not able to overlap communication with computation as the latencies to initiate the operations take longer than the operations themselves effectively serializing communication and computation. On the other hand, CPU-free execution can still achieve adequate levels of overlap even when latencies dominate [Wagner, 2020, Ismayilov et al., 2023]. Second, the parallelism offered by within-kernel communication is well-suited for persistent kernels to take advantage of. With the use of efficient communication and synchronization APIs, this execution model can achieve higher bandwidths and lower latencies than CPU-controlled implementations. Additionally, CPU-free execution, by virtue of inlining communication with computation, is well-suited for applications with fine-grained communication [Chen et al., 2022b]. Third, ROC_SHMEM allows for the first time to *fully* migrate application execution to the device with zero reliance on CPU helper threads. ROC_SHMEM, in particular, is the only fully *device-initiated*, *device-triggered* and *device-controlled* method available. NVSHMEM with its IBGDA transport can also migrate a significant amount of execution to GPU but must still rely on the CPU for functional correctness.

However, there are several challenges facing this model of execution. One major challenge is that persistent kernels can result in reduced occupancy potentially bottlenecking computation. As of today, if global device or multi-GPU barriers are required, persistent kernels must be launched in a *cooperative* manner. This means that only as many threads can be launched as can run concurrently at the same time making hardware oversubscription impossible. As a result, workload decomposition and scheduling, which were previously handled by the hardware scheduler, now need to be manually done by the programmer. This manual approach is unlikely to be as efficient as hardware-based scheduling, and compute-intensive applications are likely to suffer. Furthermore, long-running persistent kernels will consume more registers and may use shared memory as well (as in ROC_SHMEM) limiting occupancy even further. Nevertheless, we see a two-fold solution to this problem. Firstly, there is a

large amount of high bandwidth shared memory available across application execution, which can potentially nullify the performance hit caused by reduced occupancy. Secondly, we predict that GPU vendors strive more and more for greater GPU autonomy, they introduce APIs that allow for hardware oversubscription. The manual decomposition can also be handled by an optimized compiler / runtime system.

Another potential problem with NVSHMEM and ROC_SHMEM is their ease of integration into existing runtimes. Both libraries center around a *symmetric heap* and all communication buffers must be allocated collectively on the same symmetric heap by all GPUs. This symmetric allocation requires library specific allocators. Existing runtimes may find it hard to add support for NVSHMEM and ROC_SHMEM because of the symmetric memory allocation requirement.

4.4 Outlook

We now discuss what we believe to be fertile ground for future research on GPU-centric communication.

4.4.1 UCX as a Pathway for GPU-Awareness

Unified Communication X (UCX) is an open-source communication framework that abstracts over several network APIs, programming models, protocols, and implementations. The idea is to provide a set of high-level primitives while hiding the low-level implementation details behind the UCX runtime. The relevance of UCX for GPU-centric communication is that its tagged and stream APIs can be used to implement a *GPU-centric* communication layer for both ROCm and CUDA. The actual GPU-centric communication then uses the corresponding native libraries [Shamis et al., 2015, The Unified Communication X Library,].

Using UCX for realizing *GPU-centric* communication is a recent direction in the literature that has started to take hold. Perhaps the most relevant example is that of *ROCm-awareness* for MPI implementations. Much early work on *GPU-aware* MPI was done for NVIDIA GPUs using native CUDA libraries and then integrated directly into MPI runtimes. Perhaps reticent to replicate the same work to make

their runtimes ROCm-aware, most MPI implementations provide ROCm-awareness only through UCX. OpenMPI additionally provides CUDA support through UCX, besides its native integration. In non-MPI work, Choi et al. extend the UCX layer in Charm++ to provide GPU-aware communication for several programming models in the Charm++ ecosystem [Choi et al., 2021a, Choi et al., 2022].

We predict that more runtimes will gravitate toward UCX to add support for GPU-centric ocommunication. Using UCX APIs frees the programmers from relying on native vendor-specific APIs and allows adding GPU-aware communication for both ROCm and CUDA. However, there are some caveats. One is that it needs to be clarified if GPU-aware UCX can overlap communication with computation. The second caveat is that the increased generality and convenience may trade off performance. A performance comparison between GPU-awareness through native APIs and that provided through UCX would be helpful. In one work in this direction, Khorassani et al. provide a native ROCm-aware runtime for MVAPICH2, which outperforms OpenMPI with UCX on a cluster of AMD GPUs [Shafie Khorassani et al., 2021]. However, the performance difference may be due to differences in the MPI implementations, not UCX.

4.4.2 Broader GPU Autonomy

The recent proliferation of *GPU-centric* communication represents the general trend toward broader GPU autonomy. Several works, early and recent, have tried to hand the GPU the reigns of domains that have traditionally been the purview of the CPU. In an early work, Stuart et al. propose methods that allow the GPU to issue callbacks to the CPU [Stuart et al., 2010]. Silberstein et al. implement GPUfs, which allows the GPU to request files on the host CPU directly from inside a GPU kernel [Silberstein et al., 2014]. Veselý et al. implement support for invoking POSIX system calls from inside GPU kernels through changes to the Linux kernel [Veselý et al., 2018]. NVIDIA's GPUDirect Storage provides a direct data path between GPUs and storage but still relies on the CPU to orchestrate execution [Thompson and C.J., 2012]. In a recent work by NVIDIA, Qureshi et al. present BaM, the first

approach that allows GPUs to directly access storage without any CPU involvement. Experimental results show that BaM outperforms GPUDirect Storage on several workloads [Qureshi et al., 2023].

These and other works show a clear trend toward general GPU autonomy. In line with this, we expect further optimizations to GPU-centric communication. Several recent mechanisms are promising, as pointed out by Punniyamurthy et al. [Punniyamurthy et al., 2023]. First, the recent TB cluster abstraction could benefit device-side communication-computation overlap and inter-TB synchronization [Evans et al., 2022]. Second, AMD's recent cache flush instructions allow flushing the cache before initiating network communication, meaning communication no longer needs to be allocated as uncacheable [Punniyamurthy et al., 2023]. Third, recent hardware trends like fatter GPU nodes and tight GPU-NIC integration are also promising [Evans et al., 2022, AMD, 2021]. For example, the most recent iteration of NVSwitch will directly connect 256 Grace Hopper Superchips enabling direct P2P all-to-all communication at an unprecented scale.

4.4.3 Lack of Debugging Support

Efficient debugging tools are essential for productive multi-GPU programming. However, the available tools are severely lacking when it comes to communication initiated from the GPU. While NVIDIA's flagship system-level debugging tool, NSight Systems, provides a detailed view of CPU-initiated communication, it falls short in providing information on GPU-initiated communication, including Direct Load/Store P2P communication and communication induced by libraries such as NCCL and NVSHMEM. Although ComScribe can successfully monitor NCCL collective communication, it is limited to a single node and has usability concerns following the recent deprecation of nvprof, which it relies on [Soyturk et al., 2021]. We believe that introducing debugging tools capable of detecting fine-grained GPU-initiated transfers both within and across nodes is crucial for further advancements in GPU-centric communication.

4.5 Summary

In this chapter, we conduct an extensive survey of GPU-centric communication, communication mechanisms proposed in response to the deficiencies of traditional multi-GPU communication models. At a high level, these advancements reduce the CPU's involvement in the critical path of execution, give the GPU more autonomy in initiating and synchronizing communication and fix the semantic mismatch between multi-GPU communication and computation. We chart out the landscape of GPU-centric communication, summarize the main methods, and expound on their most salient features, including associated benefits and challenges.

This chapter, in part, is currently being readied for submission for publication. The thesis author is the primary investigator and author of the material.

Chapter 5

CONCLUSION

In the first part of this thesis, we propose a fully federated multi-GPU execution model and show its viability on the widely used Conjugate Gradient solver using persistent kernels, thread block specialization, device-initiated communication, and synchronization. By eliminating costly host-side communication and synchronization routines and moving the control path to devices completely, we can achieve significantly reduced communication latencies, allowing better overlap when execution time is bound by data movement across devices. Our experiments on 8 NVIDIA A100 GPUs showed that the CPU-free model could significantly improve performance, especially in communication latency-bounded scenarios.

Next, we conduct a comprehensive survey of *GPU-centric* communication. We discuss vendor mechanisms that provide primitives that reduce CPU involvement in the critical path of execution. Next, we shift gears and discuss how these low-level primitives coalesce into higher-level GPU-centric paradigms. With the apparent trend toward greater GPU autonomy, we believe that GPU-centric communication will be even more prominent in future HPC systems.

BIBLIOGRAPHY

[Afanasyev et al., 2021] Afanasyev, A., Bianco, M., Mosimann, L., Osuna, C., Thaler, F., Vogt, H., Fuhrer, O., VandeVondele, J., and Schulthess, T. C. (2021). Gridtools: A framework for portable weather and climate applications. *SoftwareX*, 15:100707.

[Agostini et al., 2017] Agostini, E., Rossetti, D., and Potluri, S. (2017). Offloading communication control logic in gpu accelerated applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 248–257, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Agostini et al., 2018] Agostini, E., Rossetti, D., and Potluri, S. (2018). Gpudirect async: Exploring gpu synchronous communication techniques for infiniband clusters. *Journal of Parallel and Distributed Computing*, 114:28–45.

[AMD,] AMD. Amd instinct mi200 instruction set architecture. `https://www.amd.com/system/files/TechDocs/instinct-mi200-cdna2-instruction-set-architecture.pdf`.

[AMD, 2021] AMD (2021). Amd cdna™ 2 architecture. `https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf`.

[AMD, 2023a] AMD (2023a). GPU-aware MPI with ROCm. `https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-gpu-aware-mpi-readme/\#`.

[AMD, 2023b] AMD (2023b). ROCK-Kernel-Driver. `https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver`.

[AMD, 2023c] AMD (2023c). ROCm Documentation: GPU-Enabled MPI. `https://rocm.docs.amd.com/en/latest/how_to/gpu_aware_mpi.html`.

[AMD, 2023d] AMD (2023d). ROCnRDMA. `https://github.com/rocmarchive/ROCnRDMA`.

[AMD, 2023e] AMD (2023e). ROC_SHMEM. `https://github.com/ROCm-Developer-Tools/ROC_SHMEM`.

[Anzt et al., 2017] Anzt, H., Gates, M., Dongarra, J., Kreutzer, M., Wellein, G., and Khler, M. (2017). Preconditioned krylov solvers on gpus. *Parallel Comput.*, 68(C):32–44.

[Awan et al., 2016] Awan, A. A., Hamidouche, K., Venkatesh, A., and Panda, D. K. (2016). Efficient large message broadcast using nccl and cuda-aware mpi for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, page 15–22, New York, NY, USA. Association for Computing Machinery.

[Awan et al., 2019] Awan, A. A., Manian, K. V., Chu, C.-H., Subramoni, H., and Panda, D. K. (2019). Optimized large-message broadcast for deep learning workloads: Mpi, mpi+nccl, or nccl2? *Parallel Computing*, 85:141–152.

[Banerjee et al., 2016] Banerjee, D. S., Hamidouche, K., and Panda, D. K. (2016). Designing high performance communication runtime for gpu managed memory: Early experiences. GPGPU '16, page 82–91, New York, NY, USA. Association for Computing Machinery.

[Bauer et al., 2014] Bauer, M., Treichler, S., and Aiken, A. (2014). Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 119–130, New York, NY, USA. Association for Computing Machinery.

[Belviranli et al., 2018] Belviranli, M. E., Lee, S., Vetter, J. S., and Bhuyan, L. N. (2018). Juggler: A dependence-aware task-based execution framework for gpus. *SIGPLAN Not.*, 53(1):54–67.

[Ben-Nun et al., 2020] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. (2020). Groute: Asynchronous multi-gpu programming model with applications to large-scale graph processing. *ACM Trans. Parallel Comput.*, 7(3).

[Bernaschi et al., 2021] Bernaschi, M., Agostini, E., and Rossetti, D. (2021). Benchmarking multi-gpu applications on modern multi-gpu integrated systems. *Concurrency and Computation: Practice and Experience*, 33(14):e5470.

[Chen et al., 2022a] Chen, C.-C., Khorassani, K. S., Anthony, Q. G., Shafi, A., Subramoni, H., and Panda, D. K. (2022a). Highly efficient alltoall and alltoallv communication algorithms for gpu systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 24–33.

[Chen et al., 2023] Chen, Y., Brock, B., Porumbescu, S., Buluc, A., Yelick, K., and Owens, J. (2023). Atos: A task-parallel gpu scheduler for graph analytics. In *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, New York, NY, USA. Association for Computing Machinery.

[Chen et al., 2022b] Chen, Y., Brock, B., Porumbescu, S., Buluç, A., Yelick, K., and Owens, J. D. (2022b). 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*, SC '22, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Choi et al., 2021a] Choi, J., Fink, Z., White, S., Bhat, N., Richards, D. F., and Kale, L. V. (2021a). Gpu-aware communication with ucx in parallel programming models: Charm++, mpi, and python. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488.

[Choi et al., 2022] Choi, J., Fink, Z., White, S., Bhat, N., Richards, D. F., and Kale, L. V. (2022). Accelerating communication for parallel programming models on gpu systems. *Parallel Computing*, 113:102969.

[Choi et al., 2021b] Choi, J., Richards, D. F., and Kale, L. V. (2021b). Charming: A scalable gpu-resident runtime system. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, page 261–262, New York, NY, USA. Association for Computing Machinery.

[Chronopoulos, 1991] Chronopoulos, A. T. (1991). Towards efficient parallel implementation of the cg method applied to a class of block tridiagonal linear systems. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 578–587, New York, NY, USA. Association for Computing Machinery.

[Chu et al., 2019] Chu, C.-H., Potluri, S., Goswami, A., Gorentla Venkata, M., Imam, N., and Newburn, C. J. (2019). Designing high-performance in-memory key-value operations with persistent gpu kernels and openshmem. In Pophale, S., Imam, N., Aderholdt, F., and Gorentla Venkata, M., editors, *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, pages 148–164, Cham. Springer International Publishing.

[Ciesko, 2023] Ciesko, J. (2023). Kokkos remote spaces repository. `https://github.com/kokkos/kokkos-remote-spaces`.

[Daoud et al., 2016] Daoud, F., Watad, A., and Silberstein, M. (2016). Gpurdma: Gpu-side library for high performance networking from gpu kernels. ROSS '16, New York, NY, USA. Association for Computing Machinery.

[Davide Rossetti, 2021] Davide Rossetti, Pak Markthub, S. H. (2021). The Latest in GPUDirect. `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32039/`.

[Dryden et al., 2018] Dryden, N., Maruyama, N., Moon, T., Benson, T., Yoo, A., Snir, M., and Van Essen, B. (2018). Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 1–13.

[Evans et al., 2022] Evans, J., Andersch, M., Sethi, V., Brito, G., and Mehta, V. (2022). Nvidia grace hopper superchip architecture in-depth. `https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/`.

[Foley and Danskin, 2017] Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17.

[Ghysels and Vanroose, 2014] Ghysels, P. and Vanroose, W. (2014). Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Comput.*, 40(7):224–238.

[Groves et al., 2020] Groves, T., Brock, B., Chen, Y., Ibrahim, K. Z., Oliker, L., Wright, N. J., Williams, S., and Yelick, K. (2020). Performance trade-offs in gpu communication: A study of host and device-initiated approaches. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 126–137.

[Gupta et al., 2012] Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Gysi et al., 2016] Gysi, T., Bär, J., and Hoefler, T. (2016). dcuda: Hardware supported overlap of computation and communication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 609–620.

[Hamidouche et al., 2016] Hamidouche, K., Awan, A. A., Venkatesh, A., and Panda, D. K. (2016). Cuda m3: Designing efficient cuda managed memory-aware mpi by exploiting gdr and ipc. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 52–61.

[Hamidouche and LeBeane, 2020] Hamidouche, K. and LeBeane, M. (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*, PPoPP '20, page 336–347, New York, NY, USA. Association for Computing Machinery.

[Hamidouche et al., 2015] Hamidouche, K., Venkatesh, A., Awan, A. A., Subramoni, H., Chu, C.-H., and Panda, D. K. (2015). Exploiting gpudirect rdma in designing high performance openshmem for nvidia gpu clusters. In *2015 IEEE International Conference on Cluster Computing*, pages 78–87.

[Harris, 2012a] Harris, M. (2012a). How to optimize data transfers in cuda c/c++. `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/`.

[Harris, 2012b] Harris, M. (2012b). How to overlap data transfers in cuda c/c++. `https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/`.

[HPE, 2021] HPE (2021). Cray MPICH Documentation. `https://cpe.ext.hpe.com/docs/mpt/mpich/intro_mpi.html`.

[Hsu et al., 2020] Hsu, C.-H., Imam, N., Langer, A., Potluri, S., and Newburn, C. J. (2020). An initial assessment of nvshmem for high performance computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–10, New York, NY, USA. Institute for Electrical and Electronics Engineers.

[Ismayilov et al., 2023] Ismayilov, I., Baydamirli, J., Sağbili, D., Wahib, M., and Unat, D. (2023). Multi-gpu communication schemes for iterative solvers: When cpus are not in charge. In *Proceedings of the 37th International Conference on Supercomputing*, ICS '23, page 192–202, New York, NY, USA. Association for Computing Machinery.

[Jeaugey, 2017] Jeaugey, S. (2017). Nccl 2.0. `https://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf`.

[Jeaugey, 2019] Jeaugey, S. (2019). Distributed neural network training: Nccl on summit. `https://www.olcf.ornl.gov/wp-content/uploads/2019/12/Summit-NCCL.pdf`.

[Karp et al., 2022] Karp, M., Jansson, N., Podobas, A., Schlatter, P., and Markidis, S. (2022). Reducing communication in the conjugate gradient method: A case study on high-order finite elements. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '22, New York, NY, USA. Association for Computing Machinery.

[Khaled Hamidouche, 2018] Khaled Hamidouche, B. B. (2018). UCX-ROCm: ROCm Integration into UCX. `https://openucx.org/wp-content/themes/jello/uploads/UCX_SC18_BOF_AMD_ROCm_UCX.pdf`.

[Klenk et al., 2014a] Klenk, B., Oden, L., and Froening, H. (2014a). Analyzing put/get apis for thread-collaborative processors. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 411–418.

[Klenk et al., 2014b] Klenk, B., Oden, L., and Fröning, H. (2014b). Gpu-centric communication for improved efficiency. In *International Workshop on Green Programming, Computing and Data Processing (GPCDP) in conjunction with International Green Computing Conference (IGCC), Dallas, TX, USA*.

[Klenk et al., 2015] Klenk, B., Oden, L., and Froning, H. (2015). Analyzing communication models for distributed thread-collaborative processors in terms of energy

and time. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 318–327.

[Kraus, 2021] Kraus, J. (2021). Multi-gpu programming models. `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31050/`.

[Langer and Dinan, 2021] Langer, A. and Dinan, J. (2021). Nvshmem: Gpu-integrated communication for nvidia gpu clusters. `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32515/`.

[lattice, 2023] lattice (2023). Quda repository. `https://github.com/lattice/quda`.

[LeBeane et al., 2017] LeBeane, M., Hamidouche, K., Benton, B., Breternitz, M., Reinhardt, S. K., and John, L. K. (2017). Gpu triggered networking for intra-kernel communications. SC '17, New York, NY, USA. Association for Computing Machinery.

[LeBeane et al., 2018] LeBeane, M., Hamidouche, K., Benton, B., Breternitz, M., Reinhardt, S. K., and John, L. K. (2018). Comp-net: Command processor networking for efficient intra-kernel communications on gpus. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA. Association for Computing Machinery.

[Li et al., 2020] Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R., and Barker, K. J. (2020). Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):94–110.

[Manian et al., 2019] Manian, K. V., Ammar, A. A., Ruhela, A., Chu, C.-H., Subramoni, H., and Panda, D. K. (2019). Characterizing cuda unified memory (um)-aware mpi designs on modern gpu architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, GPGPU '19, page 43–52, New York, NY, USA. Association for Computing Machinery.

[Maruyama et al., 2020] Maruyama, N., Essen, B. V., Ciesko, J., Wilke, J., Trott, C., Hsu, C.-H., Imam, N., Dinan, J., Langer, A., Newburn, C., and Potluri, S. (2020). Scaling scientific computing with nvshmem. `https://developer.nvidia.com/blog/scaling-scientific-computing-with-nvshmem/`.

[Meuer et al., 2023] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. (2023). Top 500. `https://www.top500.org/`. Accessed: 2023-07-29.

[Miyoshi et al., 2012] Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M., and Yoshinaga, T. (2012). Flat: A gpu programming framework to provide embedded mpi. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, page 20–29, New York, NY, USA. Association for Computing Machinery.

[MPI, 2021] MPI, I. S. (2021). IBM Spectrum MPI Version 10.2 Release Notes. `https://www.ibm.com/docs/en/smpi/10.2?topic=release-notes`.

[Muthukrishnan et al., 2021] Muthukrishnan, H., Nellans, D., Lustig, D., Fessler, J. A., and Wenisch, T. F. (2021). Efficient multi-gpu shared memory via automatic optimization of fine-grained transfers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 139–152.

[Namashivayam et al., 2023] Namashivayam, N., Kandalla, K., au2, J. B. W. I., Kaplan, L., and Pagel, M. (2023). Exploring fully offloaded gpu stream-aware message passing.

[Namashivayam et al., 2022] Namashivayam, N., Kandalla, K., White, T., Radcliffe, N., Kaplan, L., and Pagel, M. (2022). Exploring gpu stream-aware message passing using triggered operations.

[NVIDIA, 2011] NVIDIA (2011). Cuda 4.0 release notes. `https://developer.nvidia.com/cuda-toolkit-40`.

[NVIDIA, 2012] NVIDIA (2012). Nvidia gpudirect™ technology. https://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf.

[NVIDIA, 2016] NVIDIA (2016). Fast multi-gpu collectives with nccl. https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl/.

[NVIDIA, 2017a] NVIDIA (2017a). Cuda 4.1 release notes. https://developer.nvidia.com/cuda-toolkit-41-archive.

[NVIDIA, 2017b] NVIDIA (2017b). Nvidia dgx-1 with tesla v100 system architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[NVIDIA, 2021] NVIDIA (2021). Improving gpu memory oversubscription performance. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/.

[NVIDIA, 2022] NVIDIA (2022). conjugategradientmultidevicecg. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/4_CUDA_Libraries/conjugateGradientMultiDeviceCG.

[NVIDIA, 2023a] NVIDIA (2023a). Cuda programming guide release 12.2. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[NVIDIA, 2023b] NVIDIA (2023b). CUDA Runtime - Device Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html\#group__CUDART__DEVICE.

[NVIDIA, 2023c] NVIDIA (2023c). Dgx-2. https://www.nvidia.com/en-gb/data-center/dgx-2/.

[NVIDIA, 2023d] NVIDIA (2023d). GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[NVIDIA, 2023e] NVIDIA (2023e). Magnum IO GDRCopy. `https://developer.nvidia.com/gdrcopy`.

[NVIDIA, 2023f] NVIDIA (2023f). Nccl. `https://developer.nvidia.com/nccl`.

[NVIDIA, 2023g] NVIDIA (2023g). Nvidia gpudirect family. `https://developer.nvidia.com/gpudirect`.

[NVIDIA, 2023h] NVIDIA (2023h). Nvidia openshmem library (nvshmem) documentation.

[NVIDIA, 2023i] NVIDIA (2023i). Nvlink and nvswitch. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[NVIDIA, 2023j] NVIDIA (2023j). Nvshmem. `https://developer.nvidia.com/nvshmem`.

[NVIDIA, 2023k] NVIDIA (2023k). Nvshmem 2.6.0 release notes. `https://docs.nvidia.com/nvshmem/release-notes/release-260.html`.

[NVIDIA, 2023l] NVIDIA (2023l). Nvshmem 2.7.0 release notes. `https://docs.nvidia.com/nvshmem/release-notes/release-270.html\#release-270`.

[Oden and Fröning, 2013] Oden, L. and Fröning, H. (2013). Ggas: Global gpu address spaces for efficient communication in heterogeneous clusters. *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8.

[Oden et al., 2014a] Oden, L., Fröning, H., and Pfreundt, F.-J. (2014a). Infiniband-verbs on gpu: A case study of controlling an infiniband network device from the gpu. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 976–983.

[Oden et al., 2014b] Oden, L., Klenk, B., and Fröning, H. (2014b). Energy-efficient collective reduce and allreduce operations on distributed gpus. In *2014 14th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 483–492.

[Oden et al., 2014c] Oden, L., Klenk, B., and Fröning, H. (2014c). Energy-efficient stencil computations on distributed gpus using dynamic parallelism and gpu-controlled communication. In *2014 Energy Efficient Supercomputing Workshop*, pages 31–40.

[OpenMPI, 2023a] OpenMPI (2023a). Open MPI v5.0.x Documentation: CUDA. `https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/cuda.html`.

[OpenMPI, 2023b] OpenMPI (2023b). Open MPI v5.0.x Documentation: ROCm. `https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/rocm.html`.

[Pak Markthub and Howell, 2022] Pak Markthub, Jim Dinan, S. P. and Howell, S. (2022). Improving network performance of hpc systems using nvidia magnum io nvshmem and gpudirect async. `https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-an`

[Pearson, 2023] Pearson, C. (2023). Interconnect bandwidth heterogeneity on amd mi250x and infinity fabric.

[Pearson et al., 2019] Pearson, C., Dakkak, A., Hashash, S., Li, C., Chung, I.-H., Xiong, J., and Hwu, W.-M. (2019). Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 209–218, New York, NY, USA. Association for Computing Machinery.

[PETSc, 2023] PETSc (2023). Petsc faq.

[Poole et al., 2011] Poole, S. W., Hernandez, O., Kuehn, J. A., Shipman, G. M.,

Curtis, A., and Feind, K. (2011). *OpenSHMEM - Toward a Unified RMA Model*, pages 1379–1391. Springer US, Boston, MA.

[Potluri et al., 2013a] Potluri, S., Bureddy, D., Wang, H., Subramoni, H., and Panda, D. (2013a). Extending openshmem for gpu computing. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1001–1012, Boston, Massachusetts, USA. Institute for Electrical and Electronics Engineers.

[Potluri et al., 2017] Potluri, S., Goswami, A., Rossetti, D., Newburn, C., Venkata, M. G., and Imam, N. (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)*, pages 253–262.

[Potluri et al., 2018] Potluri, S., Goswami, A., Venkata, M. G., and Imam, N. (2018). Efficient breadth first search on multi-gpu systems using gpu-centric openshmem. In Gorentla Venkata, M., Imam, N., and Pophale, S., editors, *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, pages 82–96, Cham. Springer International Publishing.

[Potluri et al., 2013b] Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., and Panda, D. K. (2013b). Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *2013 42nd International Conference on Parallel Processing*, pages 80–89.

[Potluri et al., 2015] Potluri, S., Rossetti, D., Becker, D., Poole, D., Gorentla Venkata, M., Hernandez, O., Shamis, P., Lopez, M. G., Baker, M., and Poole, W. (2015). Exploring openshmem model to program gpu-based extreme-scale systems. In *Revised Selected Papers of the Second Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies - Volume 9397*, OpenSHMEM 2015, page 18–35, Berlin, Heidelberg. Springer-Verlag.

[Potluri et al., 2012] Potluri, S., Wang, H., Bureddy, D., Singh, A., Rosales, C., and Panda, D. K. (2012). Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1848–1857.

[Punniyamurthy et al., 2023] Punniyamurthy, K., Beckmann, B. M., and Hamidouche, K. (2023). Gpu-initiated fine-grained overlap of collective communication with computation.

[Qureshi et al., 2023] Qureshi, Z., Mailthody, V. S., Gelado, I., Min, S., Masood, A., Park, J., Xiong, J., Newburn, C. J., Vainbrand, D., Chung, I.-H., Garland, M., Dally, W., and Hwu, W.-m. (2023). Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 325–339, New York, NY, USA. Association for Computing Machinery.

[Rossetti et al., 2016] Rossetti, D., Potluri, S., and Fontaine, D. (2016). State of gpudirect technologies. `https://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf`.

[Schroeder, 2011] Schroeder, T. C. (2011). Peer-to-peer & unified virtual addressing. `https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf`.

[Shafie Khorassani et al., 2021] Shafie Khorassani, K., Hashmi, J., Chu, C.-H., Chen, C.-C., Subramoni, H., and Panda, D. K. (2021). Designing a rocm-aware mpi library for amd gpus: Early experiences. In Chamberlain, B. L., Varbanescu, A.-L., Ltaief, H., and Luszczek, P., editors, *High Performance Computing*, pages 118–136, Cham. Springer International Publishing.

[Shainer et al., 2011] Shainer, G., Ayoub, A., Lui, P., Liu, T., Kagan, M., Trott, C. R., Scantlen, G., and Crozier, P. S. (2011). The development of mellanox/n-

vidia gpudirect over infiniband—a new model for gpu to gpu communications. *Computer Science - Research and Development*, 26(3):267–273.

[Shamis et al., 2015] Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., et al. (2015). Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE.

[Shao et al., 2022] Shao, C., Guo, J., Wang, P., Wang, J., Li, C., and Guo, M. (2022). Oversubscribing gpu unified virtual memory: Implications and suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 67–75, New York, NY, USA. Association for Computing Machinery.

[Shi et al., 2014] Shi, R., Potluri, S., Hamidouche, K., Perkins, J., Li, M., Rossetti, D., and Panda, D. K. D. K. (2014). Designing efficient small message transfer mechanism for inter-node mpi communication on infiniband gpu clusters. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10.

[Shimokawabe et al., 2011] Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., and Matsuoka, S. (2011). Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA. Association for Computing Machinery.

[Silberstein et al., 2014] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. (2014). Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1).

[Silberstein et al., 2016] Silberstein, M., Kim, S., Huh, S., Zhang, X., Hu, Y.,

Wated, A., and Witchel, E. (2016). Gpunet: Networking abstractions for gpu programs. 34(3).

[Sourouri et al., 2014] Sourouri, M., Gillberg, T., Baden, S. B., and Cai, X. (2014). Effective multi-gpu communication using multiple cuda streams and threads. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 981–986.

[Soyturk et al., 2021] Soyturk, M. A., Akhtar, P., Tezcan, E., and Unat, D. (2021). Monitoring collective communication among gpus.

[Steinberger et al., 2014] Steinberger, M., Kenzel, M., Boechat, P., Kerbl, B., Dokter, M., and Schmalstieg, D. (2014). Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6).

[Stuart et al., 2010] Stuart, J. A., Cox, M., and Owens, J. D. (2010). Gpu-to-cpu callbacks. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, page 365–372, Berlin, Heidelberg. Springer-Verlag.

[The Unified Communication X Library,] The Unified Communication X Library. The Unified Communication X Library. `http://www.openucx.org`.

[Thompson and C.J., 2012] Thompson, A. and C.J., N. (2012). Gpudirect storage: A direct path between storage and gpu memory. `https://developer.nvidia.com/blog/gpudirect-storage/`.

[Trott, 2018] Trott, C. (2018). Early experience with nvshmem: Extending the kokkos programming model with pgas semantics. `https://www.osti.gov/servlets/purl/1806950`.

[Venkatesh et al., 2017] Venkatesh, A., Hamidouche, K., Potluri, S., Rosetti, D., Chu, C.-H., and Panda, D. K. (2017). Mpi-gds: High performance mpi designs with gpudirect-async for cpu-gpu control flow decoupling. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 151–160.

[Veselý et al., 2018] Veselý, J., Basu, A., Bhattacharjee, A., Loh, G. H., Oskin, M., and Reinhardt, S. K. (2018). Generic system calls for gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 843–856.

[Wagner, 2020] Wagner, M. (2020). Gtc 20: Overcoming latency barriers: Strong scaling hpc applications with nvshmem. `https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21673/`.

[Wahib and Maruyama, 2014] Wahib, M. and Maruyama, N. (2014). Scalable kernel fusion for memory-bound GPU applications. In *SC*, pages 191–202, New Orleans, LA, USA. IEEE Computer Society.

[Wang et al., 2019] Wang, G., Venkataraman, S., Phanishayee, A., Thelin, J., Devanur, N., and Stoica, I. (2019). Blink: Fast and generic collectives for distributed ml.

[Wang et al., 2014] Wang, H., Potluri, S., Bureddy, D., Rosales, C., and Panda, D. K. (2014). Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605.

[Wang et al., 2011a] Wang, H., Potluri, S., Luo, M., Singh, A., Sur, S., and Panda, D. (2011a). Mvapich2gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science - Research and Development*, 26:257–266.

[Wang et al., 2011b] Wang, H., Potluri, S., Luo, M., Singh, A. K., Ouyang, X., Sur, S., and Panda, D. K. (2011b). Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2. In *2011 IEEE International Conference on Cluster Computing*, pages 308–316.

[Wang et al., 2023] Wang, Y., Feng, B., Wang, Z., Geng, T., Li, A., Barker, K., and Ding, Y. (2023). Mgg: Accelerating graph neural networks with fine-

grained intra-kernel communication-computation pipelining on multi-gpu platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[Weingram et al., 2023] Weingram, A., Li, Y., Qi, H., Ng, D., Dai, L., and Lu, X. (2023). xccl: A survey of industry-led collective communication libraries for deep learning. *Journal of Computer Science and Technology*, 38(1):166–195.

[Wu et al., 2016] Wu, W., Bosilca, G., vandeVaart, R., Jeaugey, S., and Dongarra, J. (2016). Gpu-aware non-contiguous data movement in open mpi. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 231–242, New York, NY, USA. Association for Computing Machinery.

[XIE et al., 2021] XIE, C., Chen, J., Firoz, J., Li, J., Song, S. L., Barker, K., Raugas, M., and Li, A. (2021). Fast and scalable sparse triangular solver for multi-gpu based hpc architectures. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA. Association for Computing Machinery.

[Yamaguchi et al., 2017] Yamaguchi, T., Fujita, K., Ichimura, T., Hori, T., Hori, M., and Wijerathne, L. (2017). Fast finite element analysis method using multiple gpus for crustal deformation and its application to stochastic inversion analysis with geometry uncertainty. In *ICCS*, volume 108 of *Procedia Computer Science*, pages 765–775, Zürich, Switzerland. Elsevier.

[Zhang et al., 2021] Zhang, J., Brown, J., Balay, S., Faibussowitsch, J., Knepley, M., Marin, O., Mills, R. T., Munson, T., Smith, B. F., and Zampini, S. (2021). The petscsf scalable communication layer.

[Zhang et al., 2023] Zhang, L., Wahib, M., Chen, P., Meng, J., Wang, X., Endo, T., and Matsuoka, S. (2023). Perks: A locality-optimized execution model for iterative memory-bound gpu applications. In *Proceedings of the 37th International*

Conference on Supercomputing, ICS '23, page 167–179, New York, NY, USA. Association for Computing Machinery.

[Zhang et al., 2020] Zhang, L., Wahib, M., Zhang, H., and Matsuoka, S. (2020). A study of single and multi-device synchronization methods in nvidia gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 483–493, New York, NY, USA. Institute for Electrical and Electronics Engineers.