# Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge

Ismayil Ismayilov
iismayilov21@ku.edu.tr
Koç University
Turkey

Javid Baydamirli
jbaydamirli21@ku.edu.tr
Koç University
Turkey

Doğan Sağbili
dsagbili17@ku.edu.tr
Koç University
Turkey

Mohamed Wahib
mohamed.attia@riken.jp
RIKEN
Japan

Didem Unat
dunat@ku.edu.tr
Koç University
Turkey

## ABSTRACT

This paper proposes a fully autonomous execution model for multi-GPU applications that completely excludes the involvement of the CPU beyond the initial kernel launch. In a typical multi-GPU application, the host serves as the orchestrator of execution by directly launching kernels, issuing communication calls, and acting as a synchronizer for devices. We argue that this orchestration, or control flow path, causes undue overhead and can be delegated entirely to devices to improve performance in applications that require communication among peers. 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 broadly used iterative solvers, 2D/3D Jacobi stencil and Conjugate Gradient(CG). Compared to the CPU-controlled baselines, the CPU-free model can improve 3D stencil communication latency by 58.8% and provide a 1.63x speedup for CG on 8 NVIDIA A100 GPUs. The project code is available at https://github.com/ParCoreLab/CPU-Free-model.

## CCS CONCEPTS

• **Computing methodologies** → **Distributed programming languages**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → *Massively parallel systems*; *Software performance.*

## KEYWORDS

Multi-GPU, GPU-initiated communication, Persistent kernels, Iterative solvers, NVSHMEM

**ACM Reference Format:**
Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3577193.3593713



This work is licensed under a Creative Commons Attribution International 4.0 License.

*ICS '23, June 21–23, 2023, Orlando, FL, USA*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0056-9/23/06.
https://doi.org/10.1145/3577193.3593713

## 1 INTRODUCTION

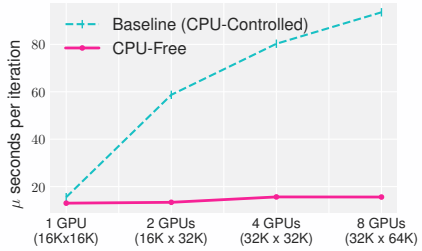
GPUs have become the leading accelerator in modern HPC systems, equipping 7 of the 10 leading Top500 supercomputers in the world [24]. As real-life applications rely on multiple GPUs to solve large problems, communication among GPUs can quickly become a performance bottleneck, leading to poor application scaling [37]. 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 [2, 18]. Advances, such as NVLink [13] and GPUDirect RDMA [18], have since allowed CUDA devices to bypass the intermediate CPU buffers during data transfers and directly communicate with their peers.

While 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. When required, the CPU also acts as a global barrier for synchronizing multiple devices. Regardless of whether the communication happens directly between devices, the presence of the CPU in the control path is implicitly assumed.

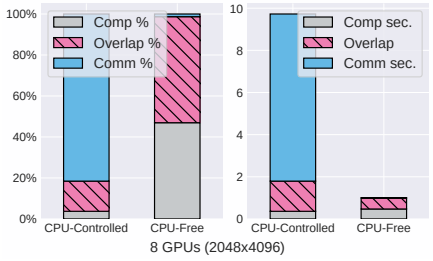
This paper argues that freeing the GPU from the host by moving the control flow to devices has several benefits for multi-GPU applications, particularly in communication-bound settings. To illustrate these advantages, we propose a distinct *CPU-Free* execution model that removes the CPU from said control 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 [15, 40]. 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

```
// 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)
}
```

**(a) Pseudo-code of CPU-controlled stencil solver [26]**




**(b) Communication overhead with no computation**




**(c) Communication overlap percentage for 2D stencil**

**Figure 1: (a) Pseudo-code of CPU-controlled 2D stencil example serving as a baseline for comparison (b) Communication and synch overheads with no computation of our CPU-free model against CPU-controlled (c) Left shows communication overlap percentages for CPU-free and CPU-controlled models. Right shows the execution times.**

data movement independently of the CPU. Finally, we hand over the task of synchronizing multiple GPUs to the devices themselves and allow them to independently synchronize with their peers. 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 two well-known iterative solvers: 2D/3D Jacobi stencils [39] and a Krylov subspace method, namely Conjugate Gradient (CG) [23]. The *CPU-free* execution model is a good fit for these types of applications as their iterative nature requires communication and computation at each iteration - tasks otherwise managed by the CPU.

Overall, our paper 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 iterative Jacobi 2D/3D and 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. Stencil kernels enjoy anywhere from 18.8% to 96.2% improvement in performance, depending on the domain size, by significantly lowering communication latencies and allowing better overlap. Standard CG and Pipelined CG variants observe 1.54x and 1.63x speedup on 18 sparse matrices compared to their CPU-controlled counterparts.

As noted, the CPU-free model's individual components have been previously proposed in other contexts. For instance, earlier works have shown the benefits of employing persistent kernels to reduce CPU involvement. A recent work by Zhang et al. [44] utilizes the GPU's shared memory and register volumes across iterations to avoid costly trips to global memory in persistent kernels. The idea of reducing CPU involvement has been studied in the literature to accelerate graph algorithms [7, 8, 35]. Other studies have proposed different methods for optimizing GPU networking by directly initiating communication from the device [2, 16, 18, 22, 29, 31, 34], as well as inter-kernel networking operations and solutions to related correctness issues [17]. However, no prior art combines these components to create a systematic way of implementing CPU-free applications. The paper serves as a practical proof of concept showing one can get significant performance improvements, especially for communication latency bounded cases, by completely cutting out the CPU and provides a blueprint for how to do this for broadly used iterative solvers.

## 2 BACKGROUND & MOTIVATION

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 [18, 33, 34], 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[13] 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.

Listing 1a illustrates the CPU's involvement in the control path with a stencil solver using the *CPU-controlled* model of execution [1]. 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 neighbors 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 1b 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 1a, 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 1c 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 [1, 42, 43]).

# 3 CPU-FREE EXECUTION MODEL

## 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 synchronization. Figure 2 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

[1]Based on multi-threaded copy overlap as provided by NVIDIA

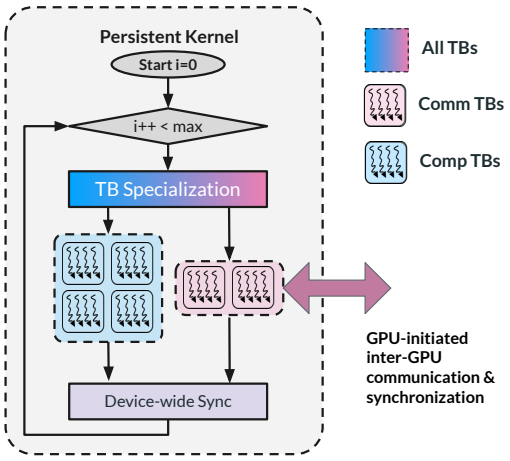



**Figure 2: CPU-free execution model leverages persistent kernels, thread-block (TB) specialization, GPU-initiated data transfers, and device-side synchronization. Comm: Communication, Comp: Computation.**

implement a long-running persistent kernel, originally proposed in [15], 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 [5]). 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 neighboring GPUs. We use NVSHMEM [27, 34], NVIDIA's implementation of OpenSHMEM [32], 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 [45], 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 (Figure 8) and **(ii)** it eliminates the frequent GPU-CPU synchronization overheads, whose latencies can quickly add up [45].

(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 (Figure 1b).

(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, [10], we observe that persistent kernels are able to achieve good overlap even when the domain size is small (as shown in Figure 1c).

(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 [44], 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. We observe the performance benefits of caching in stencil methods from maintaining a single kernel throughout the computation (Figure 5).

## 3.3 Use-Case 1: Stencil Methods

Iterative stencil solvers [41] are natural candidates for the proposed execution model as their temporal dependence across the domain requires data movement among all devices and synchronization at each iteration. Thus, we present a CPU-free implementation of multi-GPU Jacobi stencils with overlapped communication as our first use case.

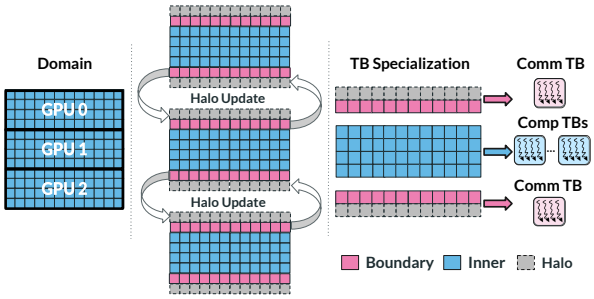



**Figure 3: Domain decomposition, halo updates, and thread block(TB) specialization for 2D5pt stencil. While communication(comm) TBs handle the boundary computation and halo update with neighbors, computation(comp) TBs handle the inner domain.**

```
__global__ void CPU_Free_Jacobi(...) {
    // Time loop on GPU
    while (iter++ < num_iterations) {
        //ⓐ Compute boundary using top neighbor's values
        if (TB_index == 0) {
            //❶ Wait for top neighbor to signal
            wait_top_neighbor(...)
            //❷ Compute top boundary using halos
            top_boundary = compute(top_halo, south, ...)
            //❸ Write to top neighbor's bottom halo
            write_top_neighbor(top_boundary)
            //❹ Signal top neighbor that iteration is done
            signal_top_neighbor(...)
        }
        //ⓑ Compute boundary with bottom neighbor's values
        if (TB_index == 1) { ... }
        //ⓒ Remaining TBs compute the inner domain
        if (TB_index == <rest>) { compute_inner() }
        //❺ Synchronize all TBs in kernel
        grid.sync()
}}
```

**Listing 1: Stencil kernel using CPU-free model**

The stencil domain can be divided equally among GPUs and split into two independent parts in each device: the inner domain and the boundary, as illustrated in Figure 3. Halo regions that refer to values in neighboring GPUs' domains are appended to each chunk and require communication at each time step. As proposed in [37], we can compute the boundary region independently of the inner domain and communicate concurrently with neighbors while the bulk of the inner domain is being processed.

The overlapping design we propose utilizes a persistent kernel that specializes a number of thread blocks to compute the boundary region and issue communication routines as shown in Listing 1. We move the time loop inside the kernel and specialize two concurrent thread blocks for the boundary region (ⓐ, ⓑ) and the remainder of the device (ⓒ) for the inner domain. Each of the boundary thread blocks ❶ initially waits on a flag for their corresponding neighbor to signal the availability of the halo region, and ❷ uses the said values to compute its boundary region. They then ❸ proceed to

commit the new values into buffers in their neighbors' memories, and ❹ signal them of their availability. ❺ Finally, all thread blocks are synchronized at the end of the loop before continuing to the next iteration.

An alternative design to specializing thread blocks in one kernel is to have two co-resident persistent kernels in separate streams, managing boundary and inner domain processing independently. This kind of configuration is more modular and easier to adapt to existing single-GPU kernels but requires an extra sync point between the local stream pairs in each GPU. However, we did not observe any significant performance improvement or degradation from this design compared to the single-stream version.

Iterative methods, as discussed earlier, benefit greatly from long-running kernels, as the intermediate results can be cached to be used in subsequent time steps instead of being committed to global memory every iteration. Recent work by Zhang et al. [44], namely PERKS, demonstrates this by explicitly caching a portion of the domain in registers and shared memory across iterations through a priority-based caching scheme. We apply our communication scheme on top of the single-GPU stencil implementation of PERKS and extend it with minimal intrusions to the upstream kernel.

### 3.4 Use-Case 2: Conjugate Gradient

To further demonstrate the broad applicability of our CPU-free execution model, we apply it 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 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 [9, 14].

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 [14, 20], 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 [14]. We refer to the traditional and pipelined variants as *Standard CG* and *Pipelined CG* respectively, and implement our design for both variants.

Figure 4 illustrates the communication/computation overlap we employ in our Pipelined CG implementation, where the global

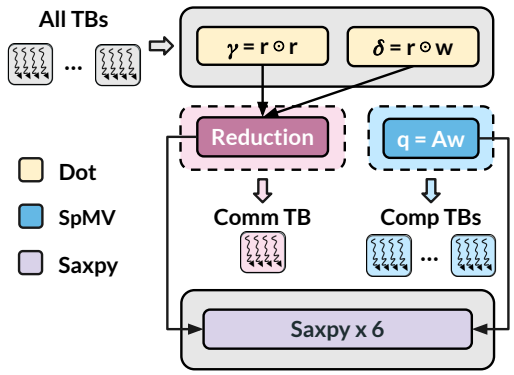



**Figure 4: Communication/computation overlap with TB specialization for Pipelined CG where reduction of dot products is overlapped with SpMV. Details omitted.**

```
__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 2: Pipelined CG kernel using CPU-free model**

reduction of dot products is overlapped with SpMV. Differently from the stencil case, 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 2 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 2, 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.

# 4 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 [17].

## 4.1 Stencil Methods

We based our stencil code on NVIDIA's open source multi-gpu programming models repository [26]. Though the samples do not necessarily have the most optimized computational kernels - the repository focuses exclusively on communication optimizations - we refer to it as our main baseline for direct comparisons of communication overhead.

*4.1.1 Synchronization and Halo Exchange.* In order to synchronize all thread blocks at the end of a given timestep, we launch our kernels using the CUDA Cooperative Groups API, which provides the device-side *grid.sync()* call. We utilize signaling operations provided by the NVSHMEM API [27] for inter-GPU synchronization. Pairs of flags are allocated in the symmetric heap for top and bottom neighbors - four in total for each processing element - using *nvshmem_malloc()*. The flags are waited on using *nvshmem_signal_wait_until()* in boundary thread blocks before computation begins to ensure neighboring devices have committed values of the previous time step. The signaling flow is essentially a semaphore wherein neighboring devices signal the availability of halos of a given time step by setting the corresponding flag to the value of the next iteration while waiting is done by comparing the flag to the current iteration. The signaling operation uses the composite *nvshmemx_putmem_signal_nbi_block()* call that performs data movement and subsequently updates a given flag on completion. This API is also used to write to halo regions residing in neighboring devices' memories. Synchronizing local concurrent kernels is done by busy waiting on a local device memory flag.

*4.1.2 Thread Block Specialization.* We specialize thread blocks to carry out inner domain computation and boundary computation / communication. In order to balance the two phases of execution, we adjust the number of thread blocks to reserve for boundary computation with the domain size. We use the following formula to determine work allocation.

$$boundary_TB_num = \frac{TB_total * boundary_size}{inner_size + 2 * boundary_size}$$

$$inner_TB_num = TB_total - 2 * boundary_TB_num$$

where $TB_total$ is the total number of thread blocks available in the device for the given thread count.

Splitting the thread blocks proportionally to the amount of work is necessary for smaller domains to achieve proper overlap, as they are susceptible to being bound by the boundary region computation and communication time otherwise.

*4.1.3 PERKS Integration.* We can apply our communication scheme to existing single-GPU kernels by swapping them into our inner-computation kernel. We extend a single-GPU stencil kernel provided by PERKS [44], as discussed in Section 3.3, as it performs explicit caching. We restrict the PERKS kernel to the inner domain while keeping it oblivious to the multi-GPU nature of execution and treating it mostly as a black box. The PERKS kernel is minimally modified to synchronize with the communication stream at the end of each iteration, and the domain resides in memory buffers shared by both. Since both kernels access the same buffers in device memory, we need to make the PERKS process a disjoint portion without memory conflicts, which is done by making it treat the boundary layers as immutable halos. As PERKS excludes read-only halos in its caching strategy, the kernel freely observes memory writes from the boundary stream by reading from the global memory in every iteration.

## 4.2 Conjugate Gradient

CUDA Samples implements a persistent kernel multi-GPU CG solver with Unified Memory [25]. 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.

*4.2.1 Synchronization.* Same as in stencils, 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()*.

*4.2.2 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.

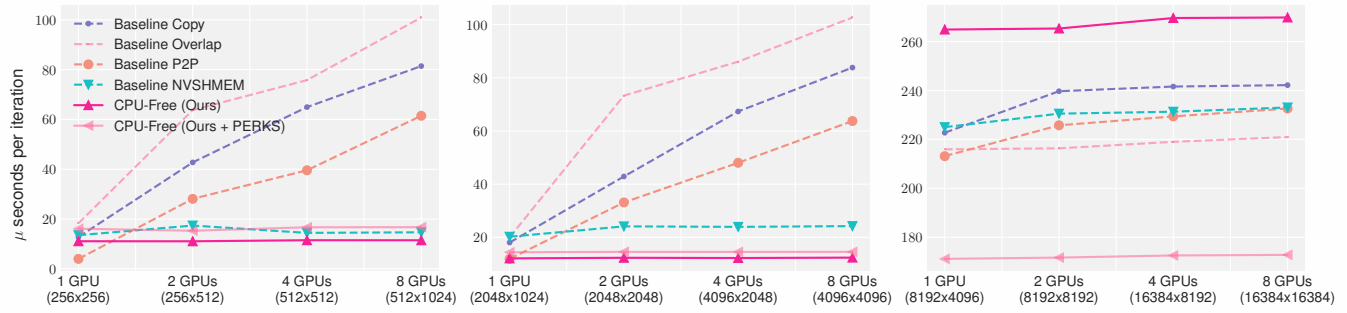



**Figure 5: Weak scaling of 2D Jacobi stencil method with small to large domain sizes on up to 8 NVIDIA A100 GPUs**

## 4.3 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. We apply such successive tiling in our implementation to assign workloads to threads, which can introduce inefficiencies compared to the hardware scheduler in large domains.

# 5 EVALUATION

This section presents the performance scaling of CPU-free execution over CPU-controlled execution for the 2D/3D Jacobi stencil and Conjugate Gradient methods. 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.

## 5.1 Stencil Benchmarks

*5.1.1 Experiment Setup and Code Variants.* We categorize the experiments into three groups of domain sizes: small, medium, and large. The domain sizes are categorized as such based on device saturation: a small domain has fewer elements than needed to keep the entirety of the device busy, while medium has sufficient elements to utilize all thread blocks, and large domains over-saturate the device. For a 2D5pt Stencil on an NVIDIA A100 with 108 SMs, we select our domains as $256^2$, $2048^2$, and $8192^2$, respectively.

The baselines are taken from NVIDIA's multi-gpu programming models repository [26], which contains Jacobi kernels with different communication schemes. We compare our implementations to the four best-performing versions:

- **Baseline Copy**: Standard CPU-controlled implementation with no explicit boundary overlap. This version only overlaps memory transfers with kernel execution using host-side asynchronous cudaMemcpy calls.
- **Baseline Overlap**: Same as above, but computes boundary rows in separate streams independently of the inner domain for explicit overlap, and synchronizes using host-side events. The explicit overlap this version performs is identical to our implementation.
- **Baseline P2P**: Communication is done through device-side direct load and stores in peer-to-peer memory. Although the communication is GPU-initiated in this version, synchronization is handled by the host.
- **Baseline NVSHMEM**: Uses device-side NVSHMEM calls for communication. This version utilizes the same family of NVSHMEM memory operations that we use in our CPU-Free implementation, except in CPU-controlled discrete kernels. It additionally uses a dedicated kernel to synchronize neighboring GPUs to avoid redundantly synchronizing all processing elements. Both kernels are launched by the CPU in every time step.
- **CPU-Free**: Our implementation as described in Section 4.1.

We additionally measure multi-GPU PERKS performance with our communication scheme, as it tiles the compute kernel to overcome the limitations of the Cooperative Groups API. Lastly, we implement a 3D7pt stencil partitioned across the $z$ axis, and adapt existing 2D baselines to 3D.

*5.1.2 Scaling Experiments.* Figure 5 shows weak-scaling measurements of per-iteration time in the aforementioned domain sizes. We vary the domain across all axes in alternating order as we double the number of devices for our weak scaling study. At 8 GPUs, we achieve speedups of 41.6% and 48.2% in small and medium domains respectively over the best-performing baseline (Baseline NVSHMEM), and 96.2% and 95.7% speedup over the fully CPU-Controlled Baseline Copy Overlap.

It should be noted that the performance degradation we experience in the largest domains compared to baseline is due to subpar tiling in the computational kernels. As per the limitations set by the cooperative groups, kernels can only be launched in configurations that guarantee the co-residency of blocks, meaning, in our case, we only allocate one block of 1024 threads on each SM. In order to process the entire domain, we tile threads in software, which causes inefficiencies in large domains. We display multi-GPU PERKS results as an alternative, as their computational kernel provides better tiling. PERKS kernel with our communication scheme

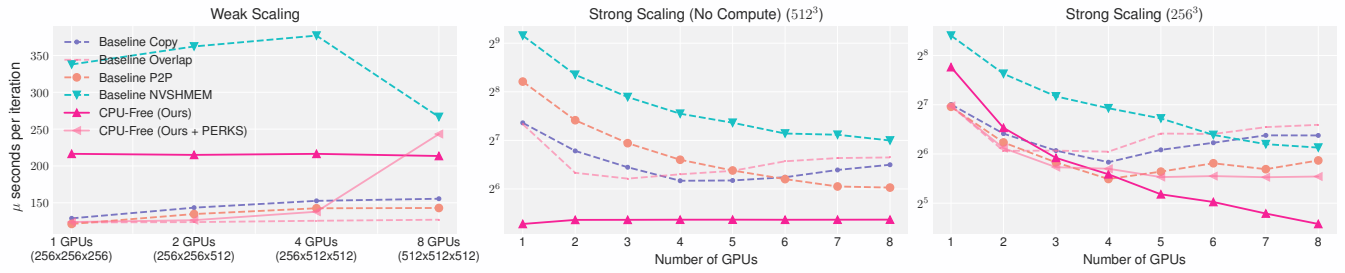



**Figure 6: 3D Jacobi stencil weak and strong scaling experiments on up to 8 NVIDIA A100 GPUs**

achieves good weak scaling within a 9% dropoff at 8 GPUs, and 18.8% speedup over the baseline on the largest domain.

Figure 6 shows 3D Jacobi performance. Though the weak scaling displays lower overall performance due to the large domain, we measure better no-compute time, which is shown in the same figure in the middle at its largest domain.

Figure 6, also conducts strong scaling experiments on a constant large 3D domain to demonstrate synchronization overheads and the overlap amount as the number of devices increases. We notice that the CPU-Free version stays largely flat, while the CPU-controlled baselines degrade. With a small number of GPUs, each GPU has a large domain that is limited by computation, not communication. However, as the GPU count increases, communication and overheads become dominant, and CPU-free clearly exhibits its communication advantages, as seen in Strong Scaling (No Compute) experiments.

In summary, CPU-free may not always be optimal, but it has proven to be highly effective for small to medium domains when CPU-induced latencies consume a substantial portion of the runtime and communication/computation can be sufficiently overlapped. Our approach excels in strong scaling scenarios where the overhead ratio increases with GPU count. Conversely, traditional CPU-controlled implementations fail to achieve overlap, causing communication and computation to be serialized in small-medium domains.

## 5.2 Conjugate Gradient

*5.2.1 Experiment Setup and Code Variants.* 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 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.

**Table 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 |

- **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 3.4.

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.

While, in practice, the CG algorithm is often coupled with a preconditioner for faster convergence [4], 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.

*5.2.2 Scaling Study.* Figure 7 shows the speedups achieved by all versions over the *Single GPU Standard* variant. Regardless of the CG

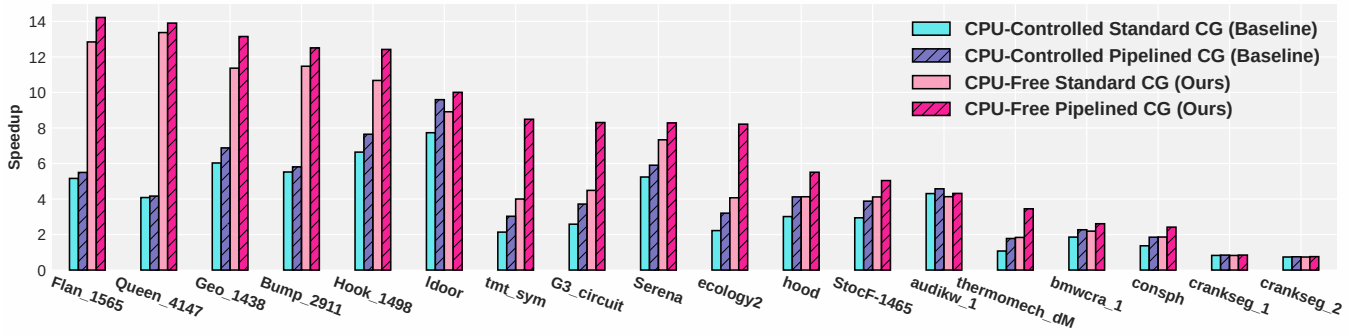



**Figure 7: Speedup over Single GPU Standard CG of various sparse matrices on 8 NVIDIA A100**

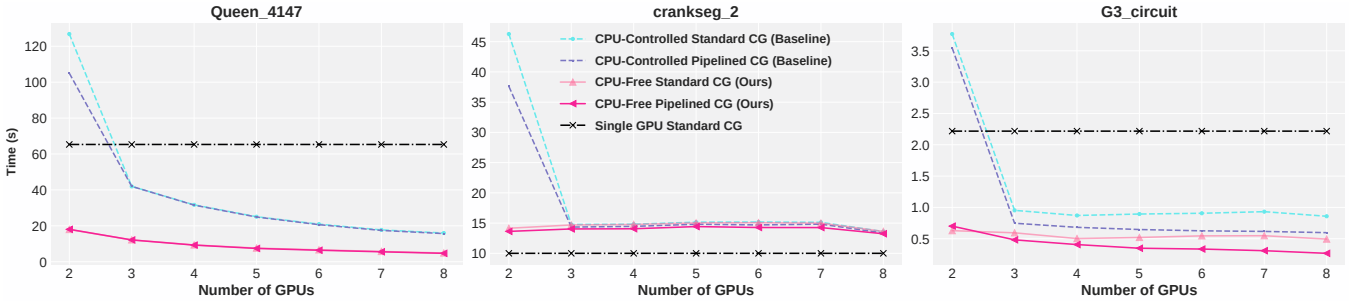



**Figure 8: Strong scaling study of CG for selected matrices with large, medium, and small sizes**

variant, CPU-free versions outperform CPU-controlled baselines on almost all matrices, but the speedup varies across matrices. As also shown in Table 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 8. 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, thermomech_dM). 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.

## 6 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. [15] 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 [40]. More recently, work by Zhang et al. [44] 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. [10] 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 [11, 30]. Other works added support for GPU networking through CPU helper threads [16, 38]. Agostini et al. [3] 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, [21] 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 [17]. 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 [19, 34–36]. 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 [28].

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. [6] 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. [7, 8] present a task-scheduling framework for irregular applications using both persistent and discrete kernels.

## 7 CONCLUSION

We propose a fully federated multi-GPU execution model and show its viability in two widely used iterative solvers: Stencil and Conjugate Gradient 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. Moreover, we can take advantage of ephemeral on-chip memory across iterations in persistent kernels. Our experiments on 8 NVIDIA A100 GPUs showed that the CPU-free model could significantly improve performance, especially in communication latency-bounded scenarios.

We view this work as an important next step toward the ultimate goal of complete GPU autonomy. In line with this, we plan to apply the CPU-free model to more applications and codify precisely when it is advantageous. We also plan to explore ways to automate porting existing CPU-controlled applications to the CPU-free model.

## ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

## REFERENCES

[1] Anton Afanasyev, Mauro Bianco, Lukas Mosimann, Carlos Osuna, Felix Thaler, Hannes Vogt, Oliver Fuhrer, Joost VandeVondele, and Thomas C. Schulthess. 2021. GridTools: A framework for portable weather and climate applications. *SoftwareX* 15 (2021), 100707.

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

[3] E. Agostini, D. Rossetti, and S. Potluri. 2018. GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters. *J. Parallel and Distrib. Comput.* 114 (2018), 28–45. https://doi.org/10.1016/j.jpdc.2017.12.007

[4] Hartwig Anzt, Mark Gates, Jack Dongarra, Moritz Kreutzer, Gerhard Wellein, and Martin Khler. 2017. Preconditioned Krylov Solvers on GPUs. *Parallel Comput.* 68, C (oct 2017), 32–44. https://doi.org/10.1016/j.parco.2017.05.006

[5] Michael Bauer, Sean Treichler, and Alex Aiken. 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* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/2555243.2555258

[6] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. *SIGPLAN Not.* 53, 1 (feb 2018), 54–67. https://doi.org/10.1145/3200691.3178492

[7] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluc, Katherine Yelick, and John Owens. 2023. Atos: A Task-Parallel GPU Scheduler for Graph Analytics. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/3545008.3545056

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

[9] A. T. Chronopoulos. 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* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 578–587. https://doi.org/10.1145/125826.126134

[10] Ching-Hsiang Chu, Sreeram Potluri, Anshuman Goswami, Manjunath Gorentla Venkata, Neena Imam, and Chris J. Newburn. 2019. Designing High-Performance In-Memory Key-Value Operations with Persistent GPU Kernels and OpenSHMEM. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, Swaroop Pophale, Neena Imam, Ferrol Aderholdt, and Manjunath Gorentla Venkata (Eds.). Springer International Publishing, Cham, 148–164.

[11] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/2931088.2931091

[12] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[13] D. Foley and J. Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[14] P. Ghysels and W. Vanroose. 2014. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. *Parallel Comput.* 40, 7 (jul 2014), 224–238. https://doi.org/10.1016/j.parco.2013.06.001

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

[16] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 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*. Institute for Electrical and Electronics Engineers, Salt Lake City, Utah, USA, 609–620. https://doi.org/10.1109/SC.2016.51

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

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

[19] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. 2020. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Institute for Electrical and Electronics Engineers, New York, NY, USA, 1–10. https://doi.org/10.1109/IPDPSW50202.2020.00104

[20] Martin Karp, Niclas Jansson, Artur Podobas, Philipp Schlatter, and Stefano Markidis. 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* (Basel, Switzerland) *(PASC '22)*.

Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/3539781.3539785

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

[22] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 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* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 29, 13 pages. https://doi.org/10.1145/3243176.3243179

[23] Jörg Liesen and Zdenek Strakos. 2012. *Krylov Subspace Methods: Principles and Analysis*. Oxford University Press, Oxford, England. https://doi.org/10.1093/acprof:oso/9780199655410.001.0001

[24] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2023. Top 500. https://www.top500.org/. Accessed: 2023-01-30.

[25] NVIDIA. 2022. conjugateGradientMultiDeviceCG. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/4_CUDA_Libraries/conjugateGradientMultiDeviceCG.

[26] NVIDIA. 2022. Multi GPU Programming Models. https://github.com/NVIDIA/multi-gpu-programming-models.

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

[28] NVIDIA. 2023. NVSHMEM 2.6.0 release notes. https://docs.nvidia.com/nvshmem/release-notes/release-260.html.

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

[30] Lena Oden, Holger Fröning, and Franz-Joseph Pfreundt. 2014. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *2014 IEEE International Parallel and Distributed Processing Symposium Workshops*. Institute for Electrical and Electronics Engineers, Phoenix, AZ, USA, 976–983. https://doi.org/10.1109/IPDPSW.2014.111

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

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

[33] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D.K. Panda. 2013. Extending OpenSHMEM for GPU Computing. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Institute for Electrical and Electronics Engineers, Boston, Massachusetts, USA, 1001–1012. https://doi.org/10.1109/IPDPS.2013.104

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

[35] Sreeram Potluri, Anshuman Goswami, Manjunath Gorentla Venkata, and Neena Imam. 2018. Efficient Breadth First Search on Multi-GPU Systems Using GPU-Centric OpenSHMEM. In *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*, Manjunath Gorentla Venkata, Neena Imam, and Swaroop Pophale (Eds.). Springer International Publishing, Cham, 82–96.

[36] Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Manjunath Gorentla Venkata, Oscar Hernandez, Pavel Shamis, M. Graham Lopez, Mathew Baker, and Wendy Poole. 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* (Annapolis, MD, USA) *(OpenSHMEM 2015)*. Springer-Verlag, Berlin, Heidelberg, 18–35. https://doi.org/10.1007/978-3-319-26428-8_2

[37] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. 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* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 3, 11 pages. https://doi.org/10.1145/2063384.2063388

[38] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking Abstractions for GPU Programs. *ACM Trans. Comput. Syst.* 34, 3, Article 9 (sep 2016), 31 pages. https://doi.org/10.1145/2963098

[39] W.F. Spotz and G.F. Carey. 1995. High-order compact finite difference methods. In *Preliminary Proceedings in International Conference on Spectral and High Order Methods*. International Conference on Spectral and High Order Methods, Houston, TX, USA, 397–408.

[40] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (nov 2014), 11 pages. https://doi.org/10.1145/2661229.2661250

[41] John C. Strikwerda. 2004. *Finite Difference Schemes and Partial Differential Equations, Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. https://doi.org/10.1137/1.9780898717938 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717938

[42] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC*. IEEE Computer Society, New Orleans, LA, USA, 191–202.

[43] Takuma Yamaguchi, Kohei Fujita, Tsuyoshi Ichimura, Takane Hori, Muneo Hori, and Lalith Wijerathne. 2017. Fast Finite Element Analysis Method Using Multiple GPUs for Crustal Deformation and its Application to Stochastic Inversion Analysis with Geometry Uncertainty. In *ICCS (Procedia Computer Science, Vol. 108)*. Elsevier, Zürich, Switzerland, 765–775.

[44] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, and Satoshi Matsuoka. 2022. Persistent Kernels for Iterative Memory-bound GPU Applications. arXiv:2204.02064 [cs.DC]

[45] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. 2020. A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Institute for Electrical and Electronics Engineers, New York, NY, USA, 483–493. https://doi.org/10.1109/IPDPS47924.2020.00057