



# Non-Blocking GPU-CPU Notifications to Enable More GPU-CPU Parallelism

Bengisu Elis

bengisu.elis@tum.de

Technische Universität München  
Garching b. München , Germany

Olga Pearce

Lawrence Livermore National  
Laboratory  
Livermore, California, USA  
pearce8@llnl.gov

David Boehme

Lawrence Livermore National  
Laboratory  
Livermore, California, USA  
boehme3@llnl.gov

Jason Burmark

Lawrence Livermore National  
Laboratory  
Livermore, California, USA  
burmark1@llnl.gov

Martin Schulz

schulzm@in.tum.de  
Technische Universität München  
Garching b. München , Germany

## ABSTRACT

GPUs are increasingly popular in HPC systems, and more applications are adopting GPUs each day. However, the control synchronization of GPUs with CPUs is suboptimal and only possible after GPU kernel termination points, resulting in serialized host and device tasks. In this paper, we propose a novel CPU-GPU notification method that enables **non-blocking in-kernel control synchronization** of device and host tasks in combination with persistent GPU kernels. Using this notification method, we increase the overlap of CPU and GPU execution and with that parallelism. We present the concept and structure of the proposed notification mechanism together with in-kernel GPU-CPU control synchronization, using halo-exchange as an example. We analyze the performance of the halo-exchange pattern using our new notification method, as well as the interference between CPU and GPU operations due to the execution overlap. Finally, we verify our results using a performance model covering the halo-exchange pattern with the new notification method.

## CCS CONCEPTS

- Computing methodologies → Parallel programming languages;
- Computer systems organization → Single instruction, multiple data;
- Software and its engineering → Software performance.

## KEYWORDS

GPU, Synchronization, Halo-exchange, MPI

### ACM Reference Format:

Bengisu Elis, Olga Pearce, David Boehme, Jason Burmark, and Martin Schulz. 2024. Non-Blocking GPU-CPU Notifications to Enable More GPU-CPU Parallelism . In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2024)*, January 25–27, 2024, Nagoya, Japan. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3635035.3635036>

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

HPCAsia 2024, January 25–27, 2024, Nagoya, Japan

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 979-8-4007-0889-3/24/01...\$15.00  
<https://doi.org/10.1145/3635035.3635036>

## 1 INTRODUCTION

As we step into the exascale age of HPC systems, GPU-accelerated systems are becoming more and more dominant. Already now, they make up over 50% of the total performance of all the systems in the Top500 list<sup>1</sup>, and this is expected to continue to rise. At the same time, due to the high performance of GPUs, more and more applications are adopting **GPUs** with the aim of offloading the **computationally intense portions** of the computation that make up the majority of the scientific applications. Usually, the **CPU** is left with seemingly smaller tasks, such as **communication, kernel launching** or **on-node synchronization** of multi GPUs [21]. However, these tasks are dependent on the **GPU-offloaded** computation tasks and, consequently, there is a need to **synchronize GPU and CPU activities**.

Device synchronizations are enabled by **device vendor APIs**, e.g., `cudaDeviceSynchronize` in CUDA. These functions enable coarse-grained synchronization at GPU kernel boundaries, serializing the execution of CPU and GPU code by blocking the CPU execution until the GPU kernel is finished. Such CPU/GPU synchronization can lead to overhead in the form of **idle CPU time** due to the **blocking synchronization, kernel termination times**, and **serialization** of the GPU and CPU tasks at the granularity of kernel boundaries.

As the GPU/CPU control synchronization leads to unavoidable overheads, there have been significant efforts to **remove the GPUs' dependency on CPUs** and with that **limiting the needed synchronization**. Some of these efforts are GPU-Direct for communication [10], Unified Virtual Memory (UVM) [2] for memory synchronization, and Cooperative Groups for enabling Multi-GPU synchronizations [8]. However, all of these improvements come with their own **overheads** and/or **require additional hardware** not available on many systems [1, 2, 7]. This is not expected to change in the future, and, consequently, we need to, even in these scenarios, find solutions that enable fine-grained CPU/GPU communication.

As completely removing the synchronization overheads between CPU and GPU is not possible, we must at least aim at performing the synchronization at finer granularity in order to decrease its impact and enable more overlap between GPU kernels and CPU tasks. This goal requires a **bi-directional notification** mechanism

<sup>1</sup><https://www.top500.org/statistics/list/>

that can synchronize GPU and CPU dependent tasks during the execution of a GPU kernel, instead of forcing the end of a kernel to achieve synchronization. However, as this notification mechanism is used at finer granularity than the traditional methods, it needs to introduce very little to **no overhead** and has to be **non-blocking** for both CPU and GPU operations.

Another advantage of using a finer-grained notification mechanism is enabling longer, **persistent GPU kernels**, which would have the potential to significantly **reduce kernel launch** and **termination overheads**. The kernel launch overhead can vary between 3 and 20  $\mu$ sec [11, 21] and with the best kernel launch overhead case, together with the kernel termination, the total overhead can add up to as high as 20  $\mu$ sec [19]. When considered individually, these overheads might seem negligible. However, typically there are many kernel launches and terminations in each iteration of an application and the overheads from multiple kernel launches and terminations can add up to significant numbers in overall application run time. In addition, **overlapping** CPU and GPU operations can **reduce the idle CPU time** spent until the GPU kernel termination when CPU only performs GPU dependent tasks.

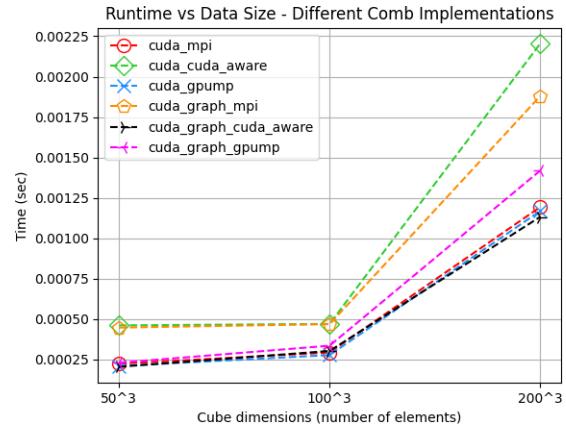
In order to achieve these goals, we design and implement a novel, yet simple **pinned host memory based notification mechanism** that enables finer granularity non-blocking communication between the CPU and the GPU without altering control flow. We showcase the performance advantages of our notification approach on the halo-exchange communication pattern using CPU triggered MPI communication. We demonstrate how enabling persistent kernels—to avoid multiple kernel launches together with reducing CPU idle time—and overlapping MPI with GPU operations improves overall performance, which more than compensates the small overhead introduced by the notification mechanism. With this use case, we also demonstrate how execution on the GPU affects the performance of concurrent MPI communication orchestrated by the CPU.

In particular, our paper makes the following contributions:

- A fine-grained notification mechanism between the GPU and the CPU to enable overlap of GPU and CPU tasks without terminating the GPU kernel (Section 3.1).
- A faster implementation of the halo-exchange communication pattern with CPU triggered MPI and our proposed notification approach (Sections 3 and 6).
- Demonstration of impact of GPU memory accesses on concurrent CPU triggered MPI communication operations (Section 4).
- An adaptation of the communication K-model to incorporate our GPU/CPU notification mechanism (Section 6.4).
- Demonstration of performance of our approach on different architectures, namely Intel, IBM and AMD CPUs with NVIDIA V100 or AMD MI100 GPUs combinations. (Section 6.3).

## 2 COMMUNICATION-RELATED LATENCIES IN ACCELERATED SIMULATIONS

Despite the immense computational parallelism and high performance of GPUs, they also expose several performance bottlenecks including kernel launch overhead and data transfer latency. Because communication to/from the GPU as well as the execution on the accelerated systems is orchestrated centrally by the CPU, it leads

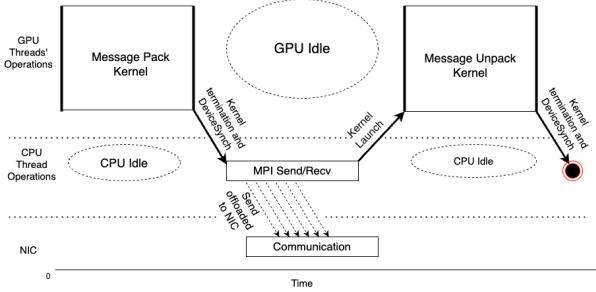


**Figure 1: Halo exchange run-time for different subdomain sizes. Performance of different communication methods combined with different GPU work scheduling methods.**

to both kernel launch overheads and data transfer latency. As a result, computationally intense applications with light and infrequent CPU/GPU communication perform well on accelerated systems, while applications with heavy or frequent communication, such as halo exchange, yield poor performance. We therefore focus on the latter application patterns and demonstrate our lightweight, non-blocking CPU-GPU notification mechanism using the widely used halo exchange pattern. In particular, we show that it enables latency hiding on the GPU by reducing the number of kernel launches and overlapping data transfers with computation.

### 2.1 The Halo Exchange Communication Pattern

Single Program Multiple Data (SPMD) is the dominant programming paradigm used in scientific simulations. The simulated domain is spatially decomposed into subdomains, and assigned to a thread or a process for computation. Periodically, the subdomain boundaries are exchanged with the logical neighbors to be used in the next iteration. This **boundary value exchange between processes working on the neighboring data subdomains** is called a **halo exchange** and is a very common communication pattern in scientific applications [3, 16, 17]. The halo exchange pattern first requires the halo values to be packed into a contiguous send buffer. With the advent of accelerated systems, the halo values typically reside in GPU memory, thus this packing is performed by the GPU, followed by the synchronization of all GPU threads together with the sender device. This is then followed by many send and receive operations by each process to exchange the halos with different neighboring processes. After the new halos are received, all GPU threads are synchronized again to start the unpack of received halo buffer to be used in the next iteration. As a consequence, each iteration of this halo-exchange pattern requires at least two CPU/GPU synchronisation points, i.e., after packing and before unpacking, to ensure the correct execution of the many communication operations between all neighbor pairs. The halo exchange thus often exhibits poor performance on accelerated systems as synchronizing the GPU with the CPU to perform this communication often comes with heavy performance penalties that occur very frequently.



**Figure 2:** CPU and GPU timeline during a halo exchange in Comb `cuda_mpi` implementation (fused kernel): Packing kernel terminates so CPU can trigger communication.

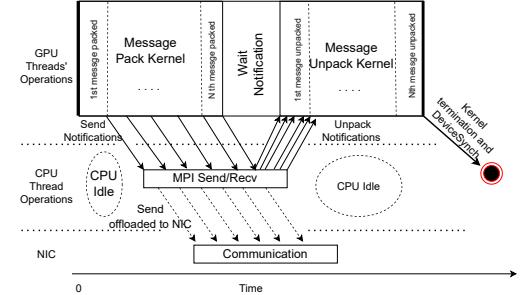
## 2.2 State of the Art and Limitations

Several solutions have been proposed to reduce the latency of data transfers to GPUs and/or to break the GPU's dependency on synchronization with CPU for communication. For inter- and intra-node GPU communication, this includes software based solutions, such as Unified Address Space (UVA) [9], CUDA-aware MPI as well as hardware-based solutions such as NVLink [14], GPUDirect RDMA, GPUDirect P2P [10]. Although the CUDA-aware MPI can utilize more advanced mechanisms and hardware than the standard MPI, in some cases standard MPI yields better performance than the CUDA-aware MPI. In their paper Hanford et al. showcase, that CUDA-aware MPI with GPUDirect can cause worse performance for message sizes above 1 MB in comparison to standard MPI [7]. Moreover, obtaining performance gains from these systems is not always easy. For example, Li et al. present via benchmarking how complicated it is to squeeze out performance from modern GPU interconnect technologies, such as NVLink [12].

## 2.3 Halo Exchange Benchmark: *Comb*

To showcase the latencies in the halo exchange, we use the *Comb* benchmark from Lawrence Livermore National Laboratory (LLNL)<sup>2</sup>. *Comb* is an open-source communication performance benchmark intended to test the performance impact of different inter-process communication methods. It simulates a configurable structured mesh halo exchange communication pattern and allows a variety of communication patterns. The benchmark can be executed with different types of parallelism, such as serial execution, OpenMP threading, CUDA streams, CUDA fused kernels or CUDA-Graphs. It supports different memory spaces, including default system allocated memory, pinned host memory, CUDA device memory and CUDA managed memory with different CUDA memory advice. For communication of the halo-region, options in *Comb* are standard MPI, CUDA-aware MPI with GPUDirect and CUDA-aware MPI with GPUDirect Asynchronous. The benchmark also allows for testing the halo-exchange communication with different data domain sizes.

To establish a baseline, we assess the current state-of-the-art for conducting multi-node GPU-to-GPU halo exchanges using *Comb* first, comparing three different communication options and two scheduling mechanisms. The three communication methods are:



**Figure 3:** GPU-CPU notification implementation: Communication starts when any buffer is ready.

- **Standard MPI (mpi):** GPU buffers are copied to CPU memory before being sent with MPI.
- **CUDA-aware MPI with GPUDirect (cuda-aware):** Standard MPI with CPU-issued send and receive calls operating directly on GPU memory buffers. On-node, GPU buffers are exchanged directly between peer GPUs. Off-node, GPU buffers are sent directly over network cards, without transferring to CPU.
- **CUDA-aware MPI w/ GPUDirect Asynchronous (gpump):** Similar to CUDA-aware MPI with GPUDirect, but GPU can resume computing right after the MPI operations are triggered.

The two scheduling mechanisms are:

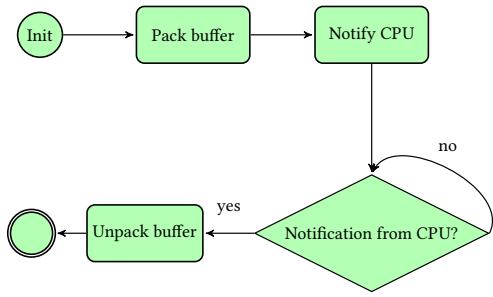
- **CUDA fused kernels (cuda):** executes halo exchange with all kernels placed on a single stream. As both pack and unpack kernel streams are fused to run on a single stream, we refer to this option as fused kernels.
- **CUDA graphs (cuda\_graph):** enables creation of complex CUDA workflows as a directed acyclic graph (DAG) where each node represents a CUDA kernel or memory operation. CUDA graphs enable more optimization and better scheduling possibilities, e.g., a define-once-run-repeatedly execution flow that reduces kernel launch overhead [20], but can be challenging to program [13].

For our evaluation, we run *Comb* on 16 nodes on a Power9+V100 system (details in Section 6.1), using 4 MPI processes on each node and 64 MPI processes in total. We study a 3D 27-point stencil halo exchange with three different halo sizes, with parameters motivated by a real-world production use case. Figure 1 shows the results. The CUDA fused kernels with standard MPI perform best because this combination minimizes the number of kernel launches. Although other combinations increase the programming complexity significantly, they don't offer any significant performance gains in comparison to the CUDA fused kernels with standard MPI combination. However, even the best performing option still has substantial overheads due to low CPU/GPU task overlap as well as due to the repeated kernel launches before and after communication. These are the issues we address with our proposed approach, laid out in the following.

## 3 NON-BLOCKING CPU/GPU NOTIFICATION

Looking at the different halo exchange implementations explored in *Comb*, the best performing implementation uses a fused kernel and standard MPI, as shown in Figure 1. This is due to the fact that a single pack kernel (and a single unpack kernel) minimizes one of

<sup>2</sup><https://github.com/LLNL/Comb>



**Figure 4: GPU Algorithm of a single halo exchange for single GPU block.**

the key GPU latencies: **kernel launch overhead**. Consider the code sequence shown as Algorithm 1 and Figure 2, which demonstrate the CPU and the GPU timelines for this implementation.

---

**Algorithm 1** Fused Kernel Comb implementation

---

```

1: for application iterations do
2:   (CPU) Launch application iteration kernel
3:   (CPU) Launch fused packing kernel
4:   (GPU) Execute application iteration
5:   (GPU) Pack all halo exchange buffers in a fused packing kernel
6:   (CPU+GPU) Synchronize the device
7:   (CPU) MPI_Send buffers to logical neighbors
8:   (CPU) MPI_Recv buffers from logical neighbors
9:   (CPU) Launch fused unpacking kernel
10:  (GPU) Unpack all halo exchange buffers in a fused unpacking kernel
11:  (CPU+GPU) Synchronize the device
12: end for
  
```

---

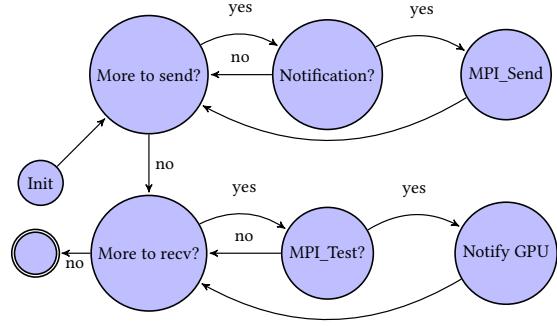
As Figure 2 and Algorithm 1 show, this implementation:

- (1) incurs the **latency of device synchronization**,
- (2) has **no overlap** between CPU and GPU activities.

In order to address these issues, we therefore must tackle both of those inefficiencies by enabling **persistent GPU kernels**, which combine **message packing** and **communication tasks** with a non-blocking CPU-GPU notification mechanism. Our proposed approach achieves this following the high-level idea demonstrated in Figure 3: as soon as the GPU has **packed any buffers**, a GPU sends a **non-blocking notification** to the CPU to let it know the buffer is ready. The CPU proceeds to send the buffer via the network interface card (NIC), while the GPU continues to pack. When all of the buffers are packed and sent, the CPU switches to checking the received messages. As soon as a message is received, the CPU notifies the GPU that the buffer is ready for unpacking. In this way, both the CPU and the GPU are able to make progress on their tasks at the same time, minimizing the synchronization overhead. Further, this scheme takes advantage of the **low kernel launch overhead** of the fused kernel, as it stays continuously active.

### 3.1 Our CPU/GPU Notification Mechanism

The notification mechanism is designed similarly to a **state machine** and a **spinlock**. For each halo message to be sent or received, there is a corresponding **notification flag** that is visible to both CPU



**Figure 5: CPU State machine for a single halo exchange.**

and GPU. Both CPU and GPU can write/read access to each flag, and by checking the state of the flag, the CPU and GPU can start performing different operations. The CPU state machine and GPU algorithm are given in Figures 4 and 5, respectively.

To implement the in-kernel CPU/GPU notification mechanism, a memory that is virtually shared by CPU and GPU is required. **UVM** enables the required accessibility and especially the zero-copy UVM memory allocation provides minimal access latency. As the notification flags are frequently accessed both from the CPU and GPU side, latency is an important performance characteristic. Therefore, the notification flags are allocated on **pinned host memory**, which offers the best latency compromise for both devices.

To read and write the notification flags, **GPU atomic operations** are used for GPU accesses. For CPU read and write accesses, **normal memory accesses** are used.

To measure the overhead of a single notification, we measure the time spent for a single read followed by a single write operation both by the CPU and GPU using the mentioned read and write access methods. The measurements on all of the available test environments (described in Section 6.1) are listed in Table 1.

In our improved halo-exchange implementation, we further **separate the notifications** between **pack/send** and **receive/unpack** to avoid collisions and contention. We achieve this using two arrays of notification flags, which we name **send-ready flags** and **unpack-ready flags**. These arrays include one notification flag for each sent or received halo message. As the first step of our workflow, the **CPU issues MPI\_Ireceives** for the new halo values computed by other GPUs. Then, it **launches the pack and unpack kernels** on different streams on the GPU, while the **CPU polls** through the send-ready flags. On the GPU side, the pack kernel packs the messages, whereas the **unpack kernel polls** to check if any messages are ready to be unpacked. Once a buffer is fully packed and ready to be sent, the **pack kernel** on the GPU **sets** the corresponding **send-ready flag**. Packing and setting the corresponding flag operations are done for all of the halo messages to be sent. On the CPU side, via the poll operations on send-flags, the CPU observes the set send-flags and **triggers** corresponding **MPI\_Isend** operations for each packed message. Upon the completion of all send operations, the CPU continues testing MPI\_Requests to check if the initially posted MPI\_Ireceives are fulfilled. For each completed receive request, the CPU sets the corresponding unpack-ready flag. Running parallel to the CPU operations, the unpack kernel can observe the

**Table 1: Single notification overhead**

Architecture	CPU overhead (nsec)	GPU overhead (nsec)
IBM Power9 + Volta100	134	236
Intel Icelake + Volta100	311	29184
AMD Rome + AMD MI100	1760	27

set of unpack-ready flags and then starts the corresponding unpack operations.

Although polling the notification flags may seem against our motivation of reducing the CPU idle time, triggering MPI communication early on and overlapping kernel execution time with MPI communication leads to a **reduction of CPU idle time**. Also, polling may seem to contradict our goal of a non-blocking notification mechanism. However, the CPU can perform other tasks between each polling iteration and doesn't have to block when there is independent CPU work that can be performed.

To ensure performance gains by the proposed implementation, it is important to investigate memory interference between CPU and GPU operations and make sure the proposed implementation doesn't lead to extra overheads and performance degradation.

In Sections 4 and 6, we focus on these two requirements. Section 4 presents investigations into interactions of CPU and GPU operations with the help of microbenchmarks, and in Section 6 we show the performance results of integrating the notification mechanism into the halo-exchange pattern. In Section 5, we explain the Comb benchmark configurations used for all of the halo-exchange communication experiments.

## 4 CPU/GPU MEMORY INTERFERENCE

Our notification mechanism allows the pack and unpack kernels to be persistent and merged together. To achieve this, we explore two different approaches: the first one is referred to as the stream approach. In this approach the pack and unpack kernels are separate and placed on two different GPU streams running in parallel, but both are persistent. The second one is referred to as the stacked approach, combining pack and unpack together in a single persistent kernel. This enables the usage of available GPU parallelism to the fullest by utilizing all available SMs and threads on each GPU.

### 4.1 Stream Approach

In the stream approach, two GPU streams are used. One of the streams is assigned with pack operations and as many GPU blocks as the number of halo messages to be packed are launched. Each GPU block is assigned one halo region to be packed into a message buffer. The pack operation is followed by one of the threads in the block notifying the CPU by updating the synchronization flag. On the second stream, running in parallel, each GPU block is assigned an unpack operation of a single message buffer and as many GPU blocks as the number of halo messages to be unpacked are launched. The kernel execution starts with polling the unpack-ready synchronization flags. Once a GPU block receives a notification from the CPU it continues to unpack the received buffer into the halo region.

This approach provides full parallelism and relies on the established stream concept to achieve concurrency between the send and the receive operations and hence is easy to implement. However, it

introduces additional scheduling overheads due to the usage of two different streams. In addition, it cannot exploit GPU parallelism to the fullest as the ratio of (utilized SMs)/(all available SMs) doesn't reach 100%. As each GPU block is assigned an MPI buffer, i.e. halo buffer, to be packed or unpacked in total 52 GPU blocks are used.

Moreover, when performing the halo-exchange of a cube shaped data domain, the size of halo regions are non-uniform, e.g., 8 corners, 12 edges and 6 faces. The messages exchanging halo corners are very small messages, whereas halo faces are very big (as shown in Table 2). In this implementation the nature of non-uniform message sizes leads to load imbalances, which is expected to harm the performance. However, the one GPU block per halo message matching is also used in the CUDA fused kernels implementation of the Comb benchmark with conventional CPU/GPU synchronization. The load imbalance due to this matching is especially problematic for performance when used without the improved synchronization mechanism and with non-persistent kernels. Without the improved synchronization mechanism it is not possible to overlap MPI with the packing of longer messages and GPU blocks, which are assigned smaller halo pack operations, idle until the pack kernel termination.

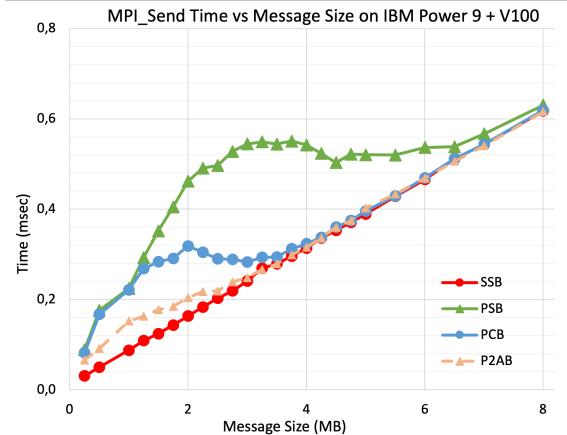
### 4.2 Stacked Kernel Approach

In this approach, the GPUs are used to the fullest capacity with maximum amount of blocks and threads available according to the architectures' limitations. The longest halo regions are packed one after the other and pack operation for each of these messages is done in parallel by all blocks and threads available. After the completion of packing of a long message, synchronization between all blocks and all threads on the GPU is done via a synchronization barrier provided by the cooperative groups mechanism. The synchronization barrier is followed by one GPU thread in one block signaling the CPU by an atomic update on the notification flag. As shorter messages are already too small to be distributed to multiple blocks these messages are packed by a single block of threads in parallel only. After each block packs their corresponding shorter messages, they notify the CPU by updating the notification flag corresponding to the message.

### 4.3 MPI and GPU Memory Access Interference

One side effect of using persistent kernels is that their execution is concurrent to CPU operations, including inter-process or inter-node messaging, which can cause interference. To study this potential impact, we developed a new micro-benchmark. It isolates behavior of simultaneous GPU memory write accesses and MPI point-to-point operations on CPU with 4 different memory proximity options for GPU and CPU buffer. These proximity options are as follows:

- **Sequential - Single Buffer (SSB):** The GPU and CPU operate on a single buffer. However, in this case the CPU and GPU operations are sequential. The CPU waits before `cudaDeviceSynchronize`

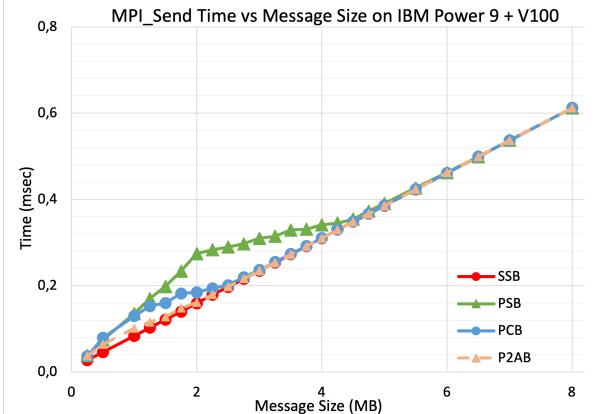


**Figure 6: MPI\_Send time for stacked implementation on Power 9 + Volta 100**

returns and GPU doesn't access the memory buffer after kernel termination. This case is a typical pattern in many applications and important as a baseline for our cases.

- **Parallel - Single Buffer (PSB):** In this case the GPU and CPU operate on a single buffer, but in this case the GPU writes into the whole buffer and then sets a notification flag on pinned host memory - similar to our notification approach. Once notified by the flag, the CPU starts MPI\_Isend operations. During the MPI operations driven by the CPU, GPU continues to write to the memory buffer. Although this is not a correct implementation, since updating the MPI buffer when it is being sent is not permitted by the MPI standard, this case serves as an extreme case to demonstrate CPU/GPU interactions.
- **Parallel - Consecutive Buffers (PCB):** In this case, CPU and GPU operations are similar to PSB and also synchronized by a notification flag. However, before the notification, the GPU writes to the MPI send buffer, and after notifying the CPU it continues writing to a consecutive memory buffer to the MPI memory region.
- **Parallel - 2MB Apart Buffers (P2AB):** This case is similar to PCB, but instead of allocating the secondary GPU access buffer consecutive to the buffer used for the MPI operations, it adds 2MB padding between the two buffers, ensuring that two separate pages are used for the two buffers.

All of the cases are tested with two different degrees of parallelism on the GPU to measure the impact of the amount of GPU parallelism on CPU operations. In the first version only 1 GPU block with the maximum allowed number of GPU threads per block is used (1024 for Volta100 and MI100) and the threads perform write accesses on each buffer element in parallel. In this version, the GPU access pattern is similar to the stream approach in Section 4.1. In the second version, as many GPU blocks and GPU threads per block are launched as the architecture limitations allow. In this version all of the GPU threads are write accessing to different elements of the buffer in parallel and within the grid synchronized by cooperative groups grid synchronization barrier. In this version, the GPU access



**Figure 7: MPI\_Send time for stream implementation on Power9 + Volta 100**

pattern is similar to the stacked approach in Section 4.2. We call these two version stream access and stacked access, respectively.

In all cases and versions, all buffers are allocated on pinned host memory and the secondary GPU memory write accesses are repeated until the MPI communication is completed. Before the launch of the measurements, we run five rounds of dummy MPI communication operations to warm up the MPI implementation. The time measurements show the time spent on MPI\_Send operations for a single buffer. The measurements are performed for different buffer sizes and use three different platforms whose details are in Section 6.1.

Figure 6 demonstrates the MPI time overheads caused by the stacked access version on Power9+Volta 100 architecture. PSB introduces significant overheads and demonstrates the impact of GPU memory accesses on MPI operations by the CPU. When message sizes are smaller than 4MB, the PCB time measurements are approximately 2.5 times more than the baseline (SSB). Although P2AB introduces some overheads, as well, it is much lower than for PCB and PSB. In Figures 10 and 8 we show that on the AMD Rome+MI100 and Icelake+Volta100 systems the impact of the GPU memory accesses has negligible to no impact on MPI\_Send time. As an orthogonal issue, we observed more volatility on the AMD platform, which we are still investigating, although, on average our conclusion stays the same.

Although stream access has worse algorithmic characteristics (explained in Section 4.1), Figures 7, 9 and 11 demonstrate that the stream access version performs at least as well as the stacked access version in terms of CPU-GPU memory interference for all systems.

## 5 COMB BENCHMARK CONFIGURATION

To emulate the halo-exchange pattern in experiments in Section 6, we use the Comb benchmark developed at LLNL, which is an extensive and complex benchmark with many parameters and measurement capabilities.

We simulate a data domain in the shape of a 3D cube. The computational data domain is uniformly decomposed into 64 sub-cubes each mapped to one of 64 MPI processes. As the computational

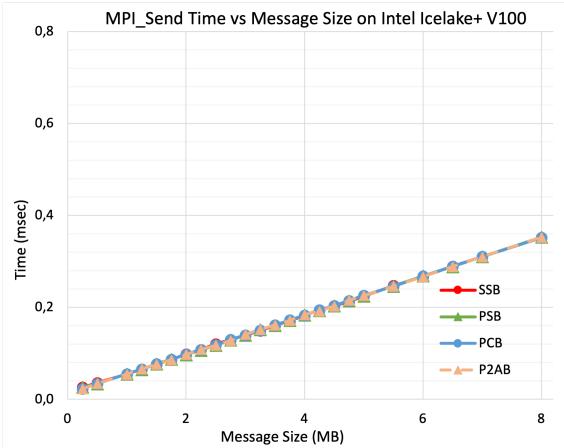


Figure 8: MPI\_Send time for stacked implementation on Intel Icelake + Volta 100

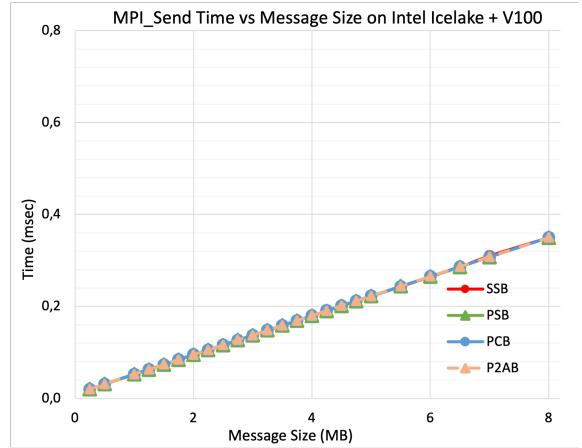


Figure 9: MPI\_Send time for stream implementation on Intel Icelake + Volta 100

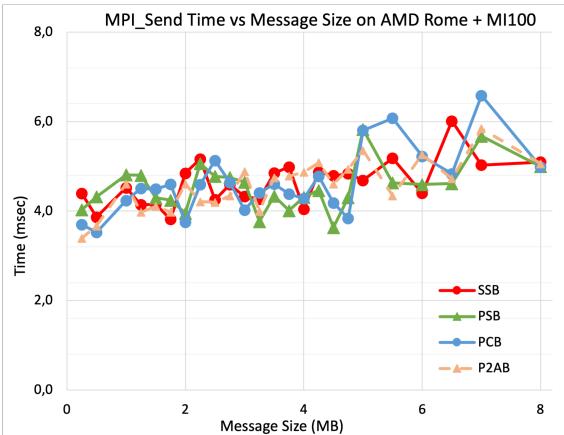


Figure 10: MPI\_Send time for stacked implementation on AMD Rome + MI100

data domain is a cube, each MPI rank has to send 26 halo messages to other MPI processes, which are assigned with the neighboring sub-cubes. Each GPU is mapped to an MPI process for communication and kernel launch. GPUs pack the halo regions of their respective sub-cube into an MPI buffer and hand the buffer to their corresponding MPI process running on the CPU. After the MPI communication, GPUs unpack the new received values from the MPI buffers into halo regions to be used in computation. For the experiments three sizes of the computational domain are used, as listed in Table 2.

Table 2: Data domain dimensions and halo region sizes used for evaluation.

Cube dimensions (data cells)	Face size (bytes)	Edge size (bytes)	Vertex size (bytes)
200 x 200 x 200	60000	1200	24
400 x 400 x 400	240000	2400	24
800 x 800 x 800	960000	4800	24

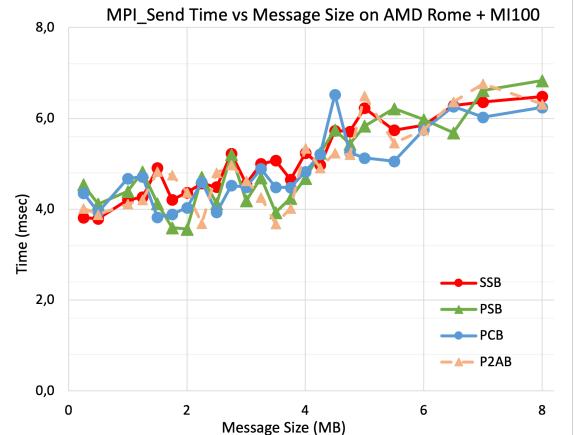


Figure 11: MPI\_Send time for stream implementation on AMD Rome + MI100

The Comb benchmark is implemented in a way that each GPU block packs and unpacks a single halo region into a single MPI buffer, which will be sent to/received from a pre-determined MPI process. This nature of the benchmark causes workload imbalance, as some blocks are only operating on the corners of their respective cubes, while some are operating on edges and or cube surfaces.

## 6 EVALUATION

In this section we present the performance measurements of the improved halo-exchange implementation with the notification system followed by a performance model of this implementation to understand the absolute quality of our approach.

### 6.1 Experimental Environment

For our experiments, we use three different systems with different architectures. The architectural variety is important to test the performance characteristics of our notification mechanism and to ascertain its performance portability.

**6.1.1 IBM Power9 + Volta 100 :** This is the largest scale test system used for our experiments. Each node has two IBM Power 9 CPUs and four Tesla V100 NVIDIA GPUs, where each Power 9 CPU contains 22 CPU cores and is connected with two GPUs by NVLink 2.0. Two GPUs, that are connected to the same CPU, are directly connected with each other by three NVLink 2.0 busses as well. This connectivity is useful for on node GPU communication with GPUDirect. However, as GPU-aware MPI is not employed in our experiments, this connectivity is not used. For our experiments we use 16 nodes of this system with CUDA version 10.1, the XL compiler version 2022.08.19 as well as Spectrum MPI 10.3.1 without any modifications or GPU aware features.

**6.1.2 Intel Icelake + Volta 100:** Our second test environment features 2 Icelake nodes with Intel Xeon Platinum 8360Y CPUs. Each node has 72 cores distributed in 2 sockets, with 1 NVIDIA Tesla V100 GPU per node. The MPI implementation is Intel MPI version 2021.6 together with CUDA 11.6 and GCC 7.5.0.

**6.1.3 AMD Rome + AMD MI100:** The third system features 2 Rome nodes with AMD EPYC 7742 CPUs, with 64 cores distributed on 2 sockets. Each node has 2 AMD MI100 GPUs. The MPI implementation is Open MPI version 5.1.0a1 with HIP version 5.4, clang version 15.0.0 and ROCm version 5.16.9.22.

## 6.2 Performance of Halo-exchange with Notification Mechanism

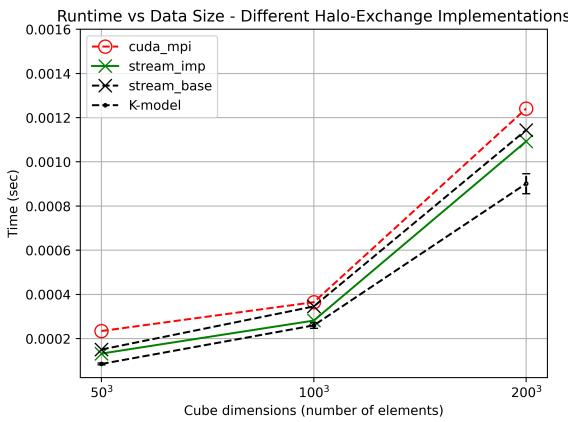
As Comb is a very complex benchmark thus it is hard to isolate and implement our improvements into the CUDA fused kernels with standard MPI implementation. Therefore we extract the communication configurations from Comb. This configuration data includes message sizes and communication peers for each MPI rank in a text file. Moreover, we produce a comparable microbenchmark as the baseline. This baseline performs the exact algorithm as Comb (as in Figure 2) and shows similar performance when executed on the same system. In addition it is easier to instrument and adopt

to the improved version in Figure 3. For the GPU parallelism, we chose the stream approach in Section 4.1, as it has the best interaction characteristics with CPU for all architectures. For this type of parallelism each GPU block is responsible of pack and unpack of a single halo buffer.

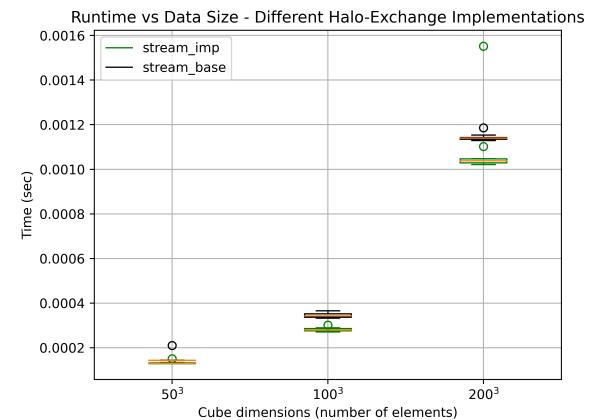
The experiments in this section are restricted to the Power9+V100 system, unless otherwise noted, as it is the largest scale system available to us. The experiments ran on 16 nodes containing 64 GPUs in total. Figure 12 shows the plots for the baseline implementation (stream\_base), the improved implementation (stream\_imp) and the Comb fused kernel implementation. Figure 13 shows the boxplots for the stream\_base and stream\_imp plots in Figure 12. These figures show, that the improved version clearly outperforms both the Comb fused kernel as well as the baseline implementations. The K-model plot reflects the modeled low bound, which is explained in Section 6.4. The latter indicates that there is still some room for theoretical improvements that even our implementation cannot exploit, yet, although it remains unclear how close an actual implementation can come to the theoretical lower bound.

## 6.3 Halo-exchange with Notification Mechanism on Different Architectures

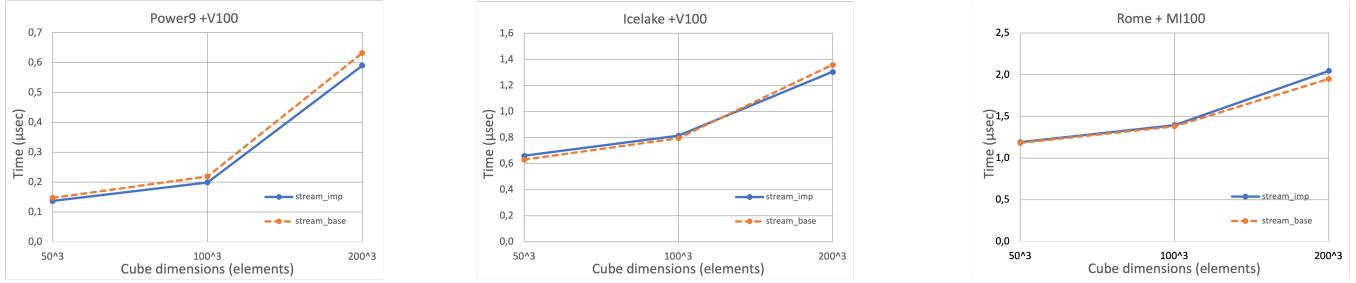
To demonstrate the performance of the halo-exchange with notification mechanism on other architectures, we used the Icelake+V100 and Rome+MI100 systems in addition to the Power9+Volta100 system. We ran the same baseline and improved notification based halo-exchange implementation in Section 6.2 by restricting the halo-exchange between two MPI processes. This is due to the fact that in the Icelake systems we have 1 GPU per node and for sake of comparability we choose 2 MPI processes. The communication configuration data is again obtained from the Comb benchmark with three different data domain sizes similar to Figure 12. As Figure 14 shows, the notification mechanism does not introduce any overheads on different architectures, although it is clear that substantial performance improvements can only be realized at scale, which is



**Figure 12: Run-time of baseline halo-exchange implementation, improved notification based halo-exchange implementation, Comb fused kernel implementation and K-model predictions with different cube sizes**



**Figure 13: Run-time of baseline halo-exchange implementation and improved notification based halo-exchange implementation with different cube sizes**



**Figure 14: Run-time vs Data size measurements of down-scaled experiments on Power9+Volta100 (top), Icelake+Volta100 (middle) and AMD Rome+MI100 (bottom) with different cube sizes**

to be expected. The measurements on Power9+Volta100 show that the improved implementation is 7% faster. On Icelake+Volta100 the improved implementation is 4% and 2% slower for smallest and mid-sized cube dimensions, whereas for largest cube dimensions it is 4% faster. On Rome+MI100 the improved implementation is 1% faster for smallest and mid-sized cube dimensions, whereas for largest cube dimensions it is 5% slower.

#### 6.4 Performance Model

As the results have shown, our approach achieves clear benefits compared to existing baselines. However, we also wanted to know how far from a lower-bound our solution is and how much unexploited optimization potential there is. For this we derived a lower bound model for our problem in two parts; GPU operations and MPI communication. The model used for the MPI communication is called K-model and it is taken from the work of Choi et al. In their paper they introduce an improved communication model for distributed GPU applications. They build their communication model on max-rate model and improve it by adding the impact of inter and intra-node communication ratio [5]. The second part of our model (GPU operations), includes linear contributions of effective pack and unpack times. Effective pack and unpack time are GPU operations which cannot be overlapped by the communication. By summing the effective pack and unpack times with the MPI time predicted by the K-model, we get the overall model and a prediction for the complete halo-exchange time. However, not all overheads are included in the K-model, turning this into a model showing a lower-bound for the achievable performance.

In order to model the GPU kernel execution and the overlap, we follow these observations: the first message to be fully packed is one of the shortest messages (edge of the halo region). Right after the pack operation of the first short message, the MPI communication starts and the rest of the pack operations are overlapped with communication. After the communication, only longer message unpack operations (face of the halo-region) influence the rest of the run-time. This is due to two reasons: a) typically short and mid sized messages are communicated much faster via MPI and the completion of sending and receiving of longer messages happens later, causing the longer messages to be unpacked last; and b) the unpack operations for short and middle sized messages are much quicker than those for long messages. Therefore, send operations

for shorter messages start earlier. Moreover, as the unpack operations of shorter messages are much faster they can be overlapped with longer message unpack operations on the GPU. For model calibration to a platform, pack and unpack time averages (Table 3) are measured by running the pack and unpack kernels with stream approach 10 times on a GPU by using different halo sizes obtained from Comb.

	Message Size (bytes)	Pack Time (μsec)	Unpack Time (μsec)
SP	24	1.75	5.54
LUP	60000	3.98	12.00
	240000	17.30	30.40
	960000	79.10	105

**Table 3: Biggest and smallest halo region sizes and their corresponding average pack and unpack times measured on Power9 + V100 system**

Considering this structure and neglecting the overheads from notification flags, we can model our improved version of halo-exchange as a summation of pack times of one short message plus MPI communication time and unpack of one long message as in Equation 1. The parameters SP and LUP are the average pack times of a short message and average unpack time of a long message.

$$SP + \{ \text{Communication Time by K-Model} \} + LUP \quad (1)$$

The K-model considers the effect of multiple processes using the network. Differing from its predecessor, the max-rate model, it considers the effect of inter and intra-node communication. Due to this update K-model can predict the communication time with better accuracy than the max-rate model and post model [5]. Choi et al. already model a halo exchange of an application. They use a machine with the same Power9+V100 architecture, and we use their architecture specific model parameters also in our work.

In the K-model the communication between different hardware units is handled separately. The model used for intra-socket messages is same as the postal model:

$$\alpha + \beta \cdot n \quad (2)$$

- $\alpha$  is the one time overhead of communication ( $= 2.92 \cdot 10^{-6}$ )
- $n$  is the message size and it varies depending on the data domain cube size and halo region - given in Table 3.
- $\beta$  is the per byte cost of communication ( $= 3.49 \cdot 10^{-11}$ )

For inter-socket and inter-node messages, the same model with different parameters is used:

$$\alpha + \frac{K_{inter}}{K_{total}} \cdot k' \cdot n \cdot \beta \quad (3)$$

The other parameters from the K-model together with their values, obtained either from Comb communication data or the work of Choi et al., are as follows:

- $K_{inter}$  is the number of messages that are sent off node. ( $= 24$ )
- $K_{total}$  is the number of total messages. ( $= 26$ )
- $k'$  is the number of processes engaged in communication per node. In our implementations it matches the number of GPUs on node. ( $= 4$ )
- $\alpha$  ( $= 3.41 \cdot 10^{-6}$  for inter socket,  $2.36 \cdot 10^{-6}$  for inter-node Eager protocol and  $1.06 \cdot 10^{-5}$  for inter-node Rendezvous protocol )
- $n$  is the message size and it varies depending on the data domain cube size and halo region - given in Table 3.
- $\beta$  in equation 3 is :

$$\frac{1}{R_{C_b} + (\frac{K_{inter}}{K_{total}} \cdot k' - 1) \cdot R_{C_i}} \quad (4)$$

- $R_{C_b}$  is the base bandwidth sustained by a single process ( $= 8.41 \cdot 10^9$  for inter socket,  $4.92 \cdot 10^9$  for inter-node Eager protocol and  $1.79 \cdot 10^{10}$  for inter-node Rendezvous protocol)
- $R_{C_i}$  is the bandwidth attainable by additional processes ( $= 7.09 \cdot 10^9$  for inter socket,  $2.31 \cdot 10^9$  for inter-node Eager protocol and  $2.22 \cdot 10^{10}$  for inter-node Rendezvous protocol)

The black dashed plot in the Figure 12 shows the lower-bound prediction from our model for different data domain sizes.

## 7 RELATED WORK

Previous work involves different control synchronization mechanisms between CPUs and GPUs, as well as performance analysis and tuning studies of GPU Communication via MPI. Although not limited to CPU/GPU control synchronization, Zhang et al. present a detailed study of available synchronization methods with different scopes and their overheads on NVIDIA GPUs [21].

Stuart et al. present a mechanism that enables CPU callbacks triggered by GPUs [18]. Their proposed approach is orthogonal to our method in-terms of the polling mechanism and usage of UVA for notification. They present an implementation with an API, but do not aim for any performance benefits. In fact their evaluation shows their approach leads to performance degradation. Whereas in our approach we kept our mechanism as lightweight as possible without structures such as freely available slots queue as in [18]. In this work we demonstrate performance improvements due to the notification mechanism in a use-case, in addition to a demonstration of the impact of GPU memory accesses on concurrent CPU triggered MPI communication.

Namashivayam et al. introduce stream-aware message passing, which has similarities to CPU queueing the MPI operations and eliminating extra kernel termination and re-launch before and after MPI operations [15]. However, in stream-aware message passing, communication operations block the GPU stream as the communication triggers and waits for the completion operations on a single GPU stream. Our approach doesn't block the GPU during communication as communication is handled by CPU. In addition, stream-aware message passing is limited to MPI only, whereas our notification mechanism has the flexibility and potential to suit any

use-case that requires CPU and GPU task overlapping with control synchronization or to enable persistent GPU kernels.

Chen et al. present their PGAS-style framework for asynchronous execution across multiple GPUs in a cluster, which also decouples the communication from synchronization [4]. They propose a task based approach for GPU programming and utilize PGAS NVSHMEM for communication. However, in our work we purposefully didn't include Openshmem implementations due to following reasons. Firstly, the most prominent Openshmem implementation NVSHMEM is not portable to all GPU platforms other than NVIDIA. Although ROCMSHMEM is still developing, portability remains to be a problem not only for different vendors but also for upcoming architectures. Secondly, Openshmem implementations are opaque and don't provide building blocks to analyze the overheads in fine resolution. Also, this opacity makes it challenging to debug when unexpected behavior is observed during execution. Lastly, the adaptation of NVSHMEM into big-scale real-life applications is challenging, as significant modifications of the computation kernels are expected and our approach assumes, that computational kernels and communication calls from GPU are independent.

Along with other valuable insights into GPU-aware communication, such as identifying the incomplete parts of some current benchmarks and challenges of developing correct MPI implementations, Hanford et al. demonstrate the performance impact of buffer locality for MPI communication [7]. They conduct experiments with GDR enabled MPI communication and with different buffer allocations, namely page-locked CPU allocation, GPU managed and GPU device allocated buffers. Their experiments are particularly interesting for our work, as they are conducted on the same architecture as ours (Power9+Volta100) and clearly demonstrate the performance benefits of GDR enabled MPI communication over standard MPI.

In addition to these methods, LeBeane et al. present an approach, that enables GPUs to trigger the NIC from within the kernel [11]. They aim to remove similar overheads as in our use-case, such as kernel start/terminate time, by restricting communication at kernel boundaries. However as their approach requires modifications to NIC hardware, it is more challenging to deploy.

## 8 CONCLUSION AND FUTURE WORK

The standard synchronization methods provided by vendor APIs are heavy weight and lead to serialization of CPU and GPU tasks. In this paper, we introduce a lightweight pinned host memory based CPU/GPU device notification mechanism that enables persistent GPU kernels, removes the need for vendor API provided synchronization and overlaps more CPU and GPU work. We investigated the performance improvement potential of the proposed notification mechanism by the use-case of halo-exchange on different architectures. Moreover, we investigated the performance effects of the choice of GPU parallelism on MPI Send operations on CPU using a microbenchmark, followed by modelling of our approach.

We model the halo-exchange implementation with our notification mechanism by extending the existing K-model. Although the notification mechanism provides performance improvements to the original implementation, the run-time measurements of the

improved halo-exchange implementation is above the model predictions. The model neglects overheads from the notification mechanism entirely and actual run-time is expected to be slightly more than the prediction. However, as one of our goals is to make the notification mechanisms lightweight, more investigation needs to be done to approach the run-time predicted by the model.

In addition, by increasing the number of notifications for longer halo message buffers, we intend to increase overlap of CPU and GPU tasks and improve the performance further for the halo-exchange use-case. Resembling partitioned communication of MPI [6], longer messages will be sent in multiple pieces without the CPU waiting the completion of the whole buffer. This approach can enable more overlap in certain cases with larger workload imbalances on GPU.

Finally, we aim to deepen our investigations about existing performance results on different architectures, together with investigating the generality of our approach using other common communication patterns in HPC benchmarks.

## ACKNOWLEDGMENTS

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Finland, Germany, Greece, France, Slovenia, Spain, and Czech Republic under grant agreement No 101093261. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-858003.

## REFERENCES

- [1] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 141–150. <https://doi.org/10.1109/IPDPS49936.2021.00023>
- [2] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) (SC’21). Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. <https://doi.org/10.1145/3458817.3480855>
- [3] Mauro Bianco. 2014. An interface for halo exchange pattern. [www.prace-ri.eu/IMG/pdf/wp86.pdf](http://www.prace-ri.eu/IMG/pdf/wp86.pdf) (2014).
- [4] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D. Owens. 2022. Scalable Irregular Parallelism with GPUs: Getting CPUs Out of the Way. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Dallas, TX, USA, 1–16. <https://doi.org/10.1109/SC41404.2022.00055>
- [5] Jaemin Choi, David F. Richards, Laxmikant V. Kale, and Abhinav Bhatele. 2020. End-to-End Performance Modeling of Distributed GPU Applications. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) (ICS’20). Association for Computing Machinery, New York, NY, USA, Article 30, 12 pages. <https://doi.org/10.1145/3392717.3392737>
- [6] Matthew G.F. Dosanjh, Andrew Worley, Derek Schafer, Prema Soundararajan, Sheikh Ghafoor, Anthony Skjellum, Purushotham V. Bangalore, and Ryan E. Grant. 2021. Implementation and evaluation of MPI 4.0 partitioned communication libraries. *Parallel Comput.* 108 (Dec 2021), 102827. <https://doi.org/10.1016/j.parco.2021.102827>
- [7] Nathan Hanford, Ramesh Pankajakshan, Edgar A. Leon, and Ian Karlin. 2020. Challenges of GPU-aware Communication in MPI. In *2020 Workshop on Exascale MPI (ExaMPI)*. IEEE, 1–10. <https://doi.org/10.1109/ExaMPI52011.2020.00006>
- [8] Mark Harris and Kyryl Pereygin. 2023. Cooperative groups: Flexible cuda thread programming. <https://developer.nvidia.com/blog/cooperative-groups/>
- [9] Feng Ji, Ashwin M. Aji, James Dinan, Darius Buntinas, Pavan Balaji, Wu-chun Feng, and Xiaosong Ma. 2012. Efficient Intranode Communication in GPU-Accelerated Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*. IEEE, Shanghai, China, 1838–1847. <https://doi.org/10.1109/IPDPSW.2012.227>
- [10] Jiri Kraus. 2022. An introduction to cuda-aware MPI. <https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/>
- [11] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. 2017. GPU Triggered Networking for Intra-Kernel Communications. (2017), 12.
- [12] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan Tallent, and Kevin Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (Jan 2020), 94–110. <https://doi.org/10.1109/TPDS.2019.2928289>
- [13] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation Using Task Graph Parallelism. In *Euro-Par 2021: Parallel Processing*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). Springer, Cham, 435–450.
- [14] Clemens Lutz, Sebastian Brefl, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 1633–1649. <https://doi.org/10.1145/3318464.3389705>
- [15] Naveen Namashivayam, Krishna Kandalla, Trey White, Nick Radcliffe, Larry Kaplan, and Mark Pagel. 2022. Exploring GPU Stream-Aware Message Passing using Triggered Operations. *arXiv:2208.04817* (Aug 2022). <http://arxiv.org/abs/2208.04817> arXiv:2208.04817 [cs].
- [16] Pier Giorgio Raponi, Fabrizio Petrin, Robert Walkup, and Fabio Checconi. 2011. Characterization of the Communication Patterns of Scientific Applications on Blue Gene/P. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops*. 1017–1024. <https://doi.org/10.1109/IPDPS.2011.249>
- [17] Lukas Spies, Amanda Bienz, David Moulton, Luke Olson, and Andrew Reisner. 2022. Tausch: A halo exchange library for large heterogeneous computing systems using MPI, OpenCL, and CUDA. *Parallel Comput.* 114 (2022), 102973. <https://doi.org/10.1016/j.parco.2022.102973>
- [18] John D. Stuart, Michael Cox, and John D. Owens. 2011. GPU-to-CPU Callbacks. In *Euro-Par 2010 Parallel Processing Workshops*, Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannataro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander (Eds.). Springer Berlin Heidelberg, 365–372.
- [19] V. Volkov and J.W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Austin, TX, 1–11. <https://doi.org/10.1109/SC.2008.5214359>
- [20] Chenle Yu, Sara Royuela, and Eduardo Quiñones. 2020. OpenMP to CUDA Graphs: A Compiler-Based Transformation to Enhance the Programmability of NVIDIA Devices. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems (SCOPES ’20)*. Association for Computing Machinery, New York, NY, USA, 42–47. <https://doi.org/10.1145/3378678.3391881>
- [21] 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)*. IEEE, New Orleans, 483–493. <https://doi.org/10.1109/IPDPS47924.2020.00057>