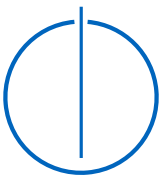
Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Design, Implementation and Test of Efficient GPU to GPU Communication Methods

Stepan Vanecek



Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Entwurf, Entwicklung und Test von Effizienten Kommunikationsmethoden zwischen GPUs

Design, Implementation and Test of Efficient GPU to GPU Communication Methods

Author:	Stepan Vanecek
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Bengisu Elis, M.Sc.
Submission Date:	Abgabetermin (15. November 2020)

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Stepan Vanecek

Acknowledgments

I would like to thank my advisor, Ms. Bengisu Elis, for her help with the project. Moreover, I would like to thank everyone who supported me, especially my family.

Abstract

Stencil codes are commonly used to solve many problems. On parallel heterogeneous systems with CPUs and GPUs, the domain is usually split and assigned to GPUs, where it is further divided to GPU blocks. The iterative distributed stencil computation consists of two steps – computation and communication, where the subdomains exchange boundary data, also called 'halo exchange'. On multi-node systems, it is crucial to efficiently transfer data from one GPU to another via MPI, as a de-facto standard solution in HPC.

In this master thesis, methods of GPU-to-GPU data exchange via MPI are examined with focus on halo exchange. The thesis describes a design of a set of naive baseline approaches and a set of optimized solutions called taskqueue. The main idea behind the taskqueue approach consists in overlapping packing and unpacking (computation) with host-to-host MPI communication, and in reusing one kernel for both packing and unpacking workloads to eliminate the kernel launch, termination, and synchronization overheads. The implementation relies on pinned host memory, a segment of main memory that is accessible by both the CPU and GPU, that the parties use to communicate. A portable solution that runs on both NVidia and AMD GPUs is designed, so that the differences on both platforms can be observed.

The performance of the taskqueue approaches is evaluated against a baseline reference on both and NVidia and AMD testbeds. The tests on NVidia yield a stable speedup that ranges from 1.09 to 1.21 for different workload sizes. Contrary to that, this approach did not prove useful on the AMD testbed, as it needed more than 200 × as much time to finish. The main reason for that are problems with concurrently reading from and writing to one memory location by the CPU and GPU.

This observation, and other observations made mainly on the AMD testbed, are identified and their implications are discussed in this work. It reveals some rigours of platform-agnostic GPU development, and discovers some unexpected behaviour patterns on the AMD GPUs combined with MPI usage. Finally, optimization to the taskqueue algorithm are proposed so that it would hopefully achieve better performance also on the AMD testbed.

Contents

1 Introduction

”The world naturally works in parallel, and so parallelism is a helpful tool for modeling the real world.” [25] ”Parallelism implies the operation of more than one task at the same time. [...] Parallel computing is the methodology involved in utilizing the concept of parallel processing and its implementation on available computer architecture, related hardware and software to solve an application problem.” [14]

Using this paradigm, it is possible to solve problems that are too large to be computed in a single stream of instructions and data. The frequency of current processors (CPUs) has not risen significantly in the last decade, as shown by K. Rupp in [30]. Therefore, the number of instructions per second that a single stream can process has not risen either. In order to solve more complex problems in a feasible amount of time, it is thus necessary to parallelize the computation in some way. Because of this fact, there is a lot of stress on parallelism as a means of improving performance of applications in the area of High-performance computing (HPC).

In [10], M. Flynn proposed a taxonomy that distinguishes two types of parallelism – data and instruction parallelism. The former describes scenarios where multiple pieces of data are processed simultaneously. When only data parallelism is present, we speak of ”Single Instruction Stream-Multiple Data Stream” (SIMD) parallelism. This class describes use-cases with a single stream of instructions, where a single instruction is applied to multiple pieces of data simultaneously. This applies mainly to vectors (for example SSE or AVX[1]) on CPUs or to Graphics Processing Unit (GPU) computing. Combining instruction and data parallelism approaches corresponds with the ”Multiple Instruction Stream-Multiple Data Stream” (MIMD) class. In this class, there are multiple streams of instruction that work on different pieces of data in parallel. Today this approach is applied at multiple levels – we can run multiple threads in a process, multiple processes on CPU cores, and we can create systems composed of multiple nodes. There are many standardized frameworks to simplify the development of MIMD applications. In the realm of HPC, the most prominent ones are OpenMP for shared memory systems and MPI for both shared and distributed memory systems. In today’s systems, applications often make use of multiple approaches to parallelism simultaneously.

1.1 GPUs and Heterogeneous architectures

GPGPUs (General Purpose Graphics Processing Units) present an extension to the classical CPU-based computing model. GPGPUs often feature thousands of threads. These threads can be used as an instrument for massive parallelism. Modern GPUs are

[1] Intel’s x86 ISA extensions for vector instructions. SSE stands for Streaming SIMD Extensions, AVX for Advanced Vector Extensions.

usually used as co-processors. Whenever the computation is parallelizable, CPUs can offload suited parts of the computation to GPUs, so that the particular subproblem is solved by a GPU in parallel and therefore faster. System architectures featuring CPUs and GPGPUs are one of the most common examples of heterogeneous system architectures. In the most recent Top500 [1] list from June 2020, 30 % of the systems use accelerators, and over 90 % of them are GPUs.

In the recent years, GPGPU computing has been increasing in importance in HPC applications, as GPUs have proven to be useful thanks to their massive SIMD parallelism capabilities. Today's high-end GPUs can outperform CPUs, while also being more power efficient thanks to more lightweight cores. "As of 2016, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10, and this ratio has been roughly constant for the past several years." [16]

However, even a GPU has a limited computing power. Most powerful GPUs nowadays reach the order of lower tens of TFLOPS. As of 2019, "the most powerful GPUs can deliver single-precision performance as high as 16 TFLOPS" [34] [2]. In order to increase the overall computing power, multiple GPUs are needed. However, GPUs still require some interaction with the host system – the CPU. One CPU can and often does serve multiple GPUs, but this number is limited. There is a lot of cooperation and communication between the parties, and large amounts of data are transferred. These actions put load on the CPU and the system bus, which has a limited bandwidth, to name a few examples. In order to avoid this problem and enable scalability beyond a couple of GPUs on top of one CPU, we usually introduce multiple nodes. Such a node consists of a CPU that operates one or a handful of GPUs. The nodes are connected with each other via a high-speed interconnect system, for example Infiniband or Omnipath. For inter-node communication, MPI is used as a de-facto standard solution.

There are two kinds of communication – intra-node, where communication between CPU, GPUs, NIC, and other devices takes place, and inter-node, where nodes communicate with other nodes. In order to utilize the computing capabilities in this topology, applications are usually written such that CPUs offload some parts of the computation to GPUs, while they still remain responsible (among other possible tasks) for the data exchange between GPUs and coordination in general. There are other approaches available which lessen the importance and role of the CPU in the communication; they will be discussed in section 2.4.

1.2 Motivation

Communication bandwidth and latency are crucial factors when designing parallel applications that do not run on systems with shared memory. The parties involved in the computation need to collaborate, which is accomplished via synchronization in form of data transfers. In the case of parallelized iterative algorithms, this procedure is repeated in each iteration, thus taking place very often. In general, a poorly performing communication scheme affects the performance of the whole application. Therefore, having an efficient

[2] NVidia Volta V100S – 16.4 TFLOPS, AMD Radeon Instinct MI60 – 14.7 TFLOPS (single-precision performance)

method of communication between the parties is extremely important for the vast majority of parallel applications running on distributed memory systems.

As mentioned previously, inter-node communication in a system can be realized via MPI. The first version of MPI [11] was released in 1994. MPI performs well in the classical CPU to CPU communication (or communication between different processes running on one CPU), which was the standard computing model when MPI was designed. Nevertheless, the standard MPI model is not well suited for using GPUs for computationally intensive tasks, which is the current trend. Adding GPUs makes the communication scheme more complex. In some implementations, the CPU is still in charge of the communication. In others, if necessary hardware support is available, GPUs can communicate directly in some cases as well. Technologies that manage this communication are presented in section 2.4.

Today, we still need CPU to CPU communication. On the top of that, the CPU needs to communicate with its GPUs. Finally, different GPUs need to synchronize data among each other as well. While the first scenario is covered by MPI and the second one is usually well optimized by vendor-specific APIs, the last one is not that straightforward. This is because neither the MPI model was designed for such heterogeneous systems, nor is it in the focus of vendor-specific APIs that aim primarily on CPU-GPU data transfer and synchronization as the standard use-case. There are existing approaches to the last type of communication as well, however they are not suitable for all kinds of host and/or device architectures, are not that performant, or easy-to-use and widespread. For this reason, the last of the three communication schemes – GPU to GPU communication – is the focus of this master thesis.

1.3 Goal

The goal of this project is to design and examine possible approaches to transferring data from one GPU to another. The focus is put on the efficiency of communication. The target use-case for which the solution will be designed is a stencil code. "A stencil is a stylized matrix computation in which a group of neighboring data elements are combined to calculate a new value. They are typically combined in the form of a sum of products." [29] Stencil code is a piece of code that performs computations on stencils. The code usually works on an iterative basis, thus the data exchange between adjacent points on the grid takes place repeatedly.

"Stencil computations are commonly used in solving partial differential equations, image processing, and geometric modelling." [29]. Solving partial differential equations is a key technique for many widely-used scenarios, such as heat transfer or fluid dynamics simulations. As stencil codes present a solution to many common problems, it is desirable to find an efficient way to solve the problem.

The exact use-case for the stencil code remains undisclosed, however a detailed description and specification of the problem this thesis examines is described in section 2.5.

In order to process stencil codes in parallel, the domain is split between the compute units (usually CPU processes or threads, or GPU blocks) so that each unit is responsible

for a fraction of the original problem. As the stencil code needs up-to-date information from adjacent cells, one tries to split the domain so that the largest possible portion of the necessary data is present on the same computing unit. However, the domain decomposition will always end up with boundaries. "Running stencil codes on distributed memory computers requires extended boundary elements to be cached and exchanged to satisfy stencil pattern locally. This elements of the sub-domain boundary are called halo. Halo exchange is often used jargon in computational sciences. It means data movement between two parallel processes holding neighbouring parts of the decomposed domain." [36]

1.4 Structure of this document

This document is organized as follows. The next chapter provides more detailed background information regarding this project – it presents GPU architectures and HIP, summarizes available technologies on efficient GPU-to-GPU communication, and gives a detailed description of the use-case. Chapter 3 describes a naive "baseline" implementation and the proposed algorithm "taskqueue" in detail. Chapter 4 presents performance results of the proposed and implemented set of solutions, and chapter 5 concludes the work and gives an outlook for future development.

2 Background

In this chapter, the technologies and research efforts related to our work are presented. First, a summary of GPGPU architecture and programming is provided. Next, the C++ runtime API and GPU kernel language HIP, that was used use for the implementation, is introduced. Then, a short introduction to MPI follows. Next section covers related research and technologies on optimizing data transfers between GPUs, commanding CPUs from GPUs, and other related areas. Finally, section 2.5 provides in-detail desctiption and definition of the use-case and the problem space.

2.1 General-Purpose Graphics Processing Units

Graphic Processing Units (GPUs) are specialized circuits that were originally created and used to accelerate graphics computations on computers. In order to manage that, lots of independent calculations have to be performed in a short time. GPUs featured an efficient parallel design in order to exploit that attribute. The capability to perform specialised operations for graphics computing in highly parallel manner started to be attractive also for other disciplines as well. Those made use of the limited capabilities and tried to exploit the massive parallelism GPUs offer. GPU vendors eventually adapted their hardware and ISAs to make GPUs easy to use also for other applications outside of the realm of graphics processing, hence the term General-Purpose GPUs. The year 2007 is an important milestone in GPGPU development as in this year, CUDA was released. It brought simplification and generalization of parallel programming on GPUs, compared to previous APIs, such as OpenGL or Direct3D, where the computation must have been expressed as work with pixels [16]. GPGPUs are the common type of GPUs currently.

Today, GPGPUs are used in a plethora of disciplines. Most prominent are use-cases for graphics processing, video rendering, gaming, cryptography, and scientific computing. In the area of scientific computing, lots of computationally-intensive application can profit from GPGPU computation. The use-cases include weather forecasting, many kinds of simulations, quantum mechanical physics modelling, and many others.

In this section, GPGPU architecture and the approaches to their programming will be briefly introduced. AMD terminology and specifications will be used, namely AMD Graphics Core Next (GCN) GPU microarchitecture and ISA. Competing NVidia products have, in general, very similar features and functionalities, often only marketed under different names. The focus is put on AMD GCN in this section as the GPUs of one of our testbeds are based on 5th generation of GCN, called "Vega 20". Moreover, as NVidia and CUDA are very popular and widely known, this section might serve as comparison for those readers who are already familiar with NVidia and CUDA concepts.

2.1.1 GPGPU Programming

The main advantage of GPGPUs compared to classical CPU programming is the massive parallelism that GPUs enable. GPUs are composed of many SIMT (Single-Instruction Multiple Threads) cores and can therefore work on many pieces of data simultaneously. Different SIMT cores can either work on larger sets of data together or run different programs. From this perspective, it can be said that GPUs are capable of MIMD parallelism.

In heterogeneous CPU-GPU systems, GPUs are used to execute computationally intensive sections of a program that can be done at least partially in parallel. When a program reaches a parallelizable subproblem, the CPU can delegate it to one or more GPUs to solve it more efficiently. In this scenario, the GPU serves as a co-processor to the CPU. When the GPU is utilized to solve a part of the program, there is a performance penalty for doing so. The CPU has the program, the data, and the resources already available in caches and registers, but none of this holds true for the GPU. Therefore, before a GPU can start computing, the particular program has to be loaded and scheduled to GPU cores, and the necessary data has to be made available as well. Classically, this is done by copying the necessary data from the main memory to the GPU memory, but there are other approaches as well. The start-up phase takes, of course, some time to finish. Therefore, offloading work to a GPU is not immediate, and this penalty has to be reckoned with. In general, problems that are not complex yet require large amounts of data movement would therefore not be suited for GPU offloading.

Vega Shader ISA [5] defines a complete application as a program that runs on the host CPU and a set of device (GPU) programs, called kernels. The host part of the program controls the program by ”commands that set GCN internal base-address and other configuration registers, specify the data domain on which the GCN GPU is to operate, invalidate and flush caches on the GCN GPU, and cause the GCN GPU to begin execution of a program.” [5]

Parallelization hierarchy

There are multiple hierarchical constructs that help programmers grasp the parallelization at different levels according to their needs. One can both distinguish each thread from each other and group and GPU's SIMT units together to work with them as a whole. The atomic unit of computation is a thread, also called work item or worker. It represents the single instruction and data stream of the device program. Figure 2.1 visualises the hierarchy, the presented terms are described in the following paragraphs.

Wavefront Threads are grouped to wavefronts (called 'warp' in CUDA terminology). There are 64 threads in a wavefront in contrast to 32 threads in a CUDA warp[1].

Wavefront can be understood as the SIMT unit of the GPU. The threads in a wavefront run in lockstep, which means the same instruction is applied to all threads of a wavefront at a time. As each thread can operate on a different set of data, wavefronts present a huge parallelization potential.

[1] When porting code between AMD and NVIDIA devices, this difference shall be taken into account.

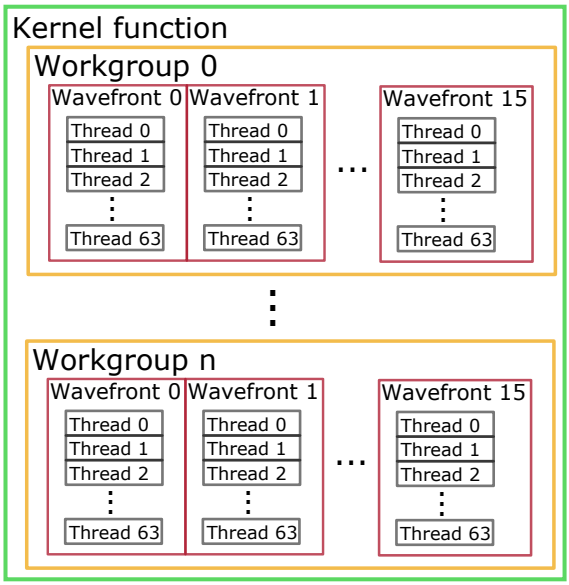


Figure 2.1: *Parallelization divisions in a GPU kernel function.*

If the device function contains conditional statements, which is not uncommon, the whole wavefront executes both *if* and *else* branches, while the instructions of one of the branches are masked off for those threads that would not enter it (do not meet the *if* condition). The only exception is when all threads fall into the same branch. Then, the instructions from the other branch are not executed at all. This actuality should be taken into consideration by the programmers as including lots of branches that can be evaluated differently by different threads may impact the performance as all branches will be executed in a serial fashion.

Workgroup Workgroups (called 'thread block' in CUDA terminology) are groups consisting of 1 to 16 wavefronts. The device function called from the host consists of a number of workgroups. The number is defined by the programmer and usually serves to split the domain. A workgroup is always scheduled (more on workload scheduling in section 2.1.2) on one compute unit ('streaming multiprocessor' in Nvidia terminology) where threads run in parallel, which means that the threads of a workgroup

- run on a GPU at the same time, and
- can share resources in the workgroup via local memory (see 2.1.1).

As there are 64 threads per wavefront and up to 16 wavefronts per workgroup, a workgroup can consist of up to 1024 threads.

GPU kernel dimensions The hierarchy of threads, wavefronts, and workgroups defines how the threads can be scheduled on the hardware, which threads will run in lockstep, or can share resources. However, the programmers do not need to specify that manually.

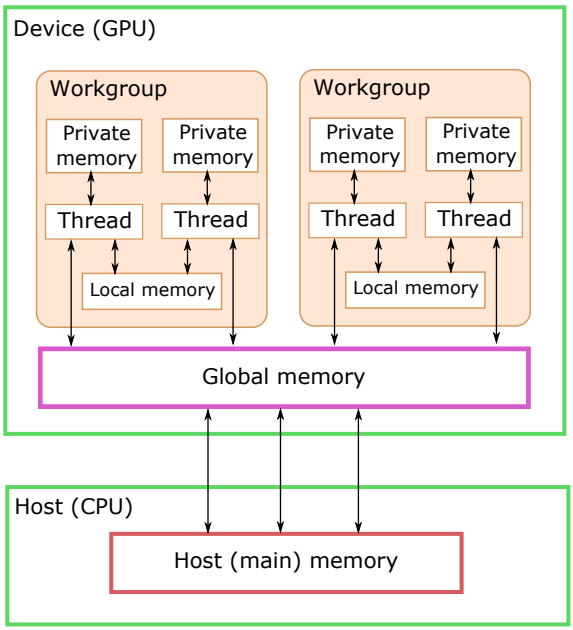


Figure 2.2: *High-Level Memory Configuration on GCN ISA. [5]*

Instead, the programmer only needs define how to split the domain in 2 ways – defining the grid and block dimensions. A problem is split into n blocks and each block is further split into m threads. Both of those are specified as 3-dimensional spaces to simplify abstracting common problems, which often also have 3 dimensions. The programmer defines number of blocks and threads in each dimension. By this, the size of the problem is specified. If 1- or 2-dimensional space is needed, the problem sizes in the third and eventually the second dimension are set to 1.

An example setup also used in our measurements is that we split the domain into $3 \times 3 \times 3 = 27$ blocks and each of those has 128 threads in one dimension ($128 \times 1 \times 1 = 128$).

Memory Management

Unlike with CPU programming, the GPU does not manage its memory automatically. That means that the programmer is, apart from writing the parallel program for the GPU itself, also responsible for memory transfers between the parties. Those can be from the host (CPU) memory to the memory of the device (GPU), the other way around, or between two devices on the same node. The first two are frequently used – from host to device as a way to provide the GPU with the data to work with and from device to host to deliver the partial result back to CPU.

From a programmer's perspective, we distinguish three kinds of memory on GPU – private memory, local memory, and global memory. They differ in their availability to different segments of the GPU. The hierarchy of these types of memory is visualised in figure 2.2.

Private memory Private memory (called 'local memory' in CUDA terminology) is the finest-grained memory type. It is only available to the thread that owns it and cannot be accessed by another thread. This memory is used for local variables that may differ from thread to thread. [26]

Local memory Local memory, also denoted Local Data Share or LDS (called 'shared memory' in CUDA terminology) is the scratchpad memory of a GPU that is shared among wavefronts (CUDA warps) of a workgroup (CUDA thread block). [26] Using local memory, the GPU can save resources for storing pieces of data that do not change for different threads in a workgroup, compared to keeping that piece of data to each thread's private memory. Moreover, threads within a workgroup can use this memory to share data and to communicate as changes made by one thread will be seen by other threads. "Local data share (LDS) is a very low-latency, RAM scratchpad for temporary data with at least one order of magnitude higher effective bandwidth than direct, uncached global memory." [5]

Global memory Global memory (also called 'global memory' in CUDA) is the third type of memory on the GPU. It is accessible for all work items of all workgroups on the whole GPU. Device memory allocation operations (`hipMalloc` for HIP or `cudaMalloc` for CUDA) allocate memory in this segment. One can think of global memory as a GPU equivalent to main memory of the CPU.

Pinned host memory Pinned host memory, also called page-locked or zero-copy memory, is yet another kind of memory accessible from the GPU, however it does not reside on the GPU itself. Pinned host memory is a segment of the host (main) memory that can be directly accessed by the device via a bus, usually PCI express (alternatively NVLink for NVidia products). Classical host memory is typically pageable, meaning the host OS kernel can move the physical data on the main memory, for example it can be offloaded to HDD. Contrary to that, page-locked memory stays on main memory and is not moved by the CPU. Therefore, accesses to the page-locked memory segments from the GPU, which are direct and do not need to involve the CPU, can be performed. The size of page-locked memory is, however, limited, so allocating too much memory of this type could lead to performance issues, as it is used by the GCN ISA for other purposes as well, for example for `hipMemcpy` memory copy in HIP (see section 2.2). Moreover, page-locked memory is used not only by GPUs. "MPI uses pinned pages for communicating over interconnect" [19].

Accesses to data allocated on pinned host memory (assuming the data is not cached, which is our case) are slower compared to all previous types as the data needs to go over the system bus. On the other hand, both host and devices can read from and write to the data located in pinned host memory. Accessing data directly on the pinned host memory is called zero-copy access.

Pinned host memory is used not only for direct accesses by the GPU as described above. When data between the host and device is copied, transferring data from or to a pinned host memory segment is faster compared to a pageable segment of main memory as the GPUs can access it directly. Therefore, it might also be beneficial to put frequently copied

data on the pinned host memory as well. Nevertheless, as all memory transfers between the host and device go through this segment, allocating too much data on pinned host memory can decrease the performance of other memory copy operations between host and devices drastically.

It is possible to choose between 2 coherency options for pinned host memory in HIP:

- **Coherent memory** allows fine-grained synchronization, such as atomic operations. As the name suggests, the data accesses remain coherent across multiple devices. In order to achieve this, the memory is not cached, meaning the data needs to be loaded from and stored back to the main memory each time, which presents a performance penalty. [2]
- **Non-coherent memory** is more efficient. It can be cached and therefore repeated load/store operations will be much faster. On the other hand, it cannot be synchronized while the kernel is running. As the data gets stored and accessed from caches, coherency is not guaranteed as the segments of data in the cache or main memory might not be the most recent ones. [2]

2.1.2 GPGPU Architecture

The focus of GPGPUs, unlike CPUs, is mainly on parallel computation. They are designed to be the slave of the host system and to take over computation that was assigned to it. GPUs can traditionally deal very well with arithmetic operations on integers. More recently, the support of floating point units (both half-, single-, and double-precision) was improved and now is comparable to that of CPUs [16]. As GPUs have a very narrow focus on computation, their compute units are usually simpler compared to CPUs. It does not need to run an OS, it does not need to connect peripherals, switch contexts, and many other duties a CPU has. This makes GPU cores more lightweitght, which means that the FLOP/mm^2 is higher and thus GPUs are easier to manufacture and more power-efficient. Thanks to these characteristics, their price per FLOPS is lower.

Workload scheduling

A GPU consists of a Command Processor and one or more Shader Engines, in case of our AMD MI-50 GPU, there are four of them. The shader engine is composed of a Workload Manager and a set of Compute Units (CUs). (There are 15 CUs per shader engine on MI-50). Scheduling starts with the command processor reading a command package from command queues (one or more per GPU). The command processor splits the command package in workgroups, and distributes them to the workload managers. Then, a workload manager further splits the workgroup and creates wavefronts for the CUs and distribute the workgroups to the compute units. [7]

User-defined blocks (wavefronts) get scheduled on compute units. All threads of a workgroup run on one CU at the same time. The local memory (LDS) is attached to each CU, so that the threads in a CU can share data, which maps to the programmer's perspective discussed in 2.1.1.

Memory hierarchy

Similarly to CPUs, GPUs have multiple options for storing the data in hardware that differ in their capacity and speed of access. The fastest and also the smallest one are the register files. Between register files and global memory, there are L1 caches and an L2 cache. The L1 cache is private for each CU, while L2 cache is global. The cache line size is 64 B [5], however for NVIDIA devices, it is 128 B [23] (the size is stored in variable `warpSize` in HIP). Finally, the global memory is the largest memory type that is also the slowest to access. "In a typical arrangement, registers for the various processing elements pull data from a set of L1 caches, which in turn access a unified, on-chip L2 cache. The L2 cache then provides high-bandwidth, low-latency access to the GPU's dedicated video memory." [5]

For each CU, the register file consists of scalar general-purpose registers (GPR) and vector GPRs. Vector GPRs contain the data for CU's vector SIMT units, while scalar registers contain data common to all threads in a wavefront, such as pointer arithmetic or constant data. These pieces of data are used by CU's scalar unit.

Accesses to pinned host memory are the slowest ones, as they need to go though the system bus. Depending on coherency settings, it is or is not cached, as describen in section 2.1.1.

Connectivity

The GPU is connected to its global memory, for our AMD MI-50, it is 32 GB HBM (High Bandwidth Memory). Next, there are PCIe controllers ensuring connectivity to the host. Finally, there are also DMA engines there, which are capable of directly accessing host's main memory. They are responsible for asynchronous memory copies between the host and the device, and possibly between multiple devices. [7]

2.2 HIP

Heterogeneous-Computing Interface for Portability (HIP) [37] is a C++ runtime API and C++-based kernel language. It is designed to ease transferring from vendor-locked CUDA applications to a portable code. CUDA is a programming language and API used for programming NVidia devices. Hence, programming with CUDA comes with vendor lock-in to NVidia hardware. NVidia has been a dominant player on the GPU market. According to [24], the market share of NVidia in the segment of discrete GPUs has been approximately between 60 % and 75 % in 2018. The rest of the market belongs almost entirely to AMD. In GPU sales overall, Intel is the market leader thanks to its dominance in integrated GPU segment. The dominance of NVidia in the discrete GPU segment is one of the reasons why CUDA programs are so widespread and vice-versa. The other reason is that CUDA has presented an efficient and simple to understand programming model when it was released in 2007 and it took years before other products caught up.

HIP is a part of AMD ROCm (Radeon Open Compute) platform that contains software projects focussing on HPC programming. ROCm is an open source modular platform that enables its users to install, on top of the core components, only the components that

CUDA	HIP
`my_kernel<<<grid, block>>> (args..)`	`hipLaunchKernelGGL(my_kernel, grid, block, LDS_size, stream, args..)`
`cudaMalloc`	`hipMalloc`
`cudaMallocHost`	`hipHostMalloc`
`cudaMemcpy`	`hipMemcpy`
`cudaStreamCreate`	`hipStreamCreate`
`cudaDeviceSynchronize`	`hipDeviceSynchronize`

Table 2.1: *Comparison of some frequently-used CUDA and HIP API calls.*

they intend to use. Version 1.0 was released in 2016. As the whole platform is not so mature yet, it is still rapidly evolving, as of autumn 2020.

We chose HIP for implementing our solutions mainly thanks to its capability of compiling code for both NVidia and AMD devices. This means that a single version of code can be run and evaluated on both platforms. This way, the code that was produced is fully portable and therefore measurements on both NVidia and AMD GPUs over the same piece of source code can be made. The experiments and benchmarks performed in [35] suggest that HIP has a comparable performance to competing C++ runtime APIs & GPU kernel languages. This will be presented in section 2.2.3 in more detail.

2.2.1 HIP and CUDA

HIP accepts the popularity of CUDA and presents an alternative, yet as similar as possible, solution. Its syntax is very similar to the one of CUDA. Table 2.1 presents some frequently-used API calls for comparison. Often, the only difference is the `cuda` or `hip` prefix. The complete list of CUDA runtime APU functions supported by HIP can be found in [27]. Unlike CUDA, HIP is open-source (released under MIT license) and the code should be compilable both for NVidia and AMD devices, hence "Interface for Portability".

HIP comes with functionality called 'HIPify'. This program should ease the transition from CUDA to HIP code, called 'hipification'. There are two flavours of this program – `hipify-perl` and `hipify-clang`. The former is a rather simple program based on string replacement. It mainly converts CUDA API calls and data types to HIP syntax. The latter is more sophisticated and tries to resolve dependencies as well. Nevertheless, neither of the programs is intended to work fully independently. Human supervision and interaction is always necessary, maybe with the exception of very trivial use-cases. From our experience with `hipify-perl`, manual code checking and corrections of the code are necessary; the tool does not do much beyond what could be done by a series of Find&Replace operations.

2.2.2 Compiling HIP code

As mentioned above, HIP source code can be compiled for both NVidia and AMD devices. As those have different ISAs, the compiled binaries, unlike the source code, are different.

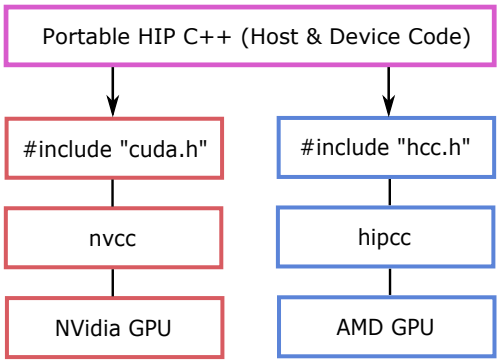


Figure 2.3: *Compiling HIP code for AMD and NVidia devices. [7]*

The binaries are vendor-specific, even microarchitecture-specific, depending on which device the programmer compiles the code for.

To enable this, there are two underlying compilers for HIP code – HIP-NVCC and HIP-Clang – each compiling for one of the vendors. As figure 2.3 shows, different platform-specific components get included when compiling for each platform. `hip_nvcc` is the program for compiling for NVidia devices. In brief, HIP header files are capable of translating HIP code to CUDA-like code that can be compiled by NVCC, which is NVidia's closed-source compiler for CUDA code. Contrary to that, HIP-Clang builds up on LLVM/Clang compiler and enables compiling HIP source code for AMD devices. Another alternative used to be `hip-hcc` compiler, using the `hcc` compiler under the hood, however it was deprecated in release 3.5 [37] in summer 2020.

`hipcc` is the overarching compiler for HIP that utilizes one of the underlying compilers – either HIP-NVCC or HIP-Clang. The programmer can either let `hipcc` detect for which vendor it should compile or manually define it in variable `HIP_PLATFORM` with `nvcc` for NVidia and `hcc` for AMD. As `hipcc` builds on other compilers, the programmer can utilize the functionality of these compilers, including compiler flags etc. However, NVCC reads different flags than Clang, so the flags for compilation with `hipcc` will often be different when compiling for an AMD device compared to compiling for an NVidia device.

2.2.3 Alternatives to HIP

There is a plethora of GPU programming languages that could be alternatives to HIP and a thorough analysis of those is beyond the scope of this section. All in all, the choice of programming language is not crucial for this project. A choice is sufficient as long as it performs well, is easy to use, works on both NVidia and AMD GPUs, is under active development and there is a large community behind the project.

ROCm platform supports multiple programming languages, such as OpenMP, OpenCL, and HIP. The AMD System Runtime (ROCr) is built on HSA Runtime API which enables executing programs written in those programming languages on AMD hardware and a host system with Linux OS. [3] Alternatives outside of ROCm platform include NVidia's CUDA,

more recently OpenCL's extension SYCL, Intel's heterogeneous computing HPC approach DPC++ (Data Parallel C++) and many others. As our focus lies, apart from NVidia, also on AMD devices, the compatibility with this vendor is a must. This requirement rules out CUDA, the most prominent programming language for NVidia GPUs.

CUDA CUDA is a GPU programming framework developed by NVidia for their devices. It is compiled with NVCC, which is NVidia's proprietary compiler.

OpenMP "ROCm offers support for offloading compute to AMD GPUs in multi-node deployments via the OpenMP application programming interface using pragma target offload directives." [4]

HC++ HC++ is a programming language, similar to C++ AMP by Microsoft. It was deprecated in favour of HIP recently, therefore, no further development of this project is to be expected.

OpenCL OpenCL is a heterogeneous programming framework that can be used to program both NVidia and AMD devices. It defines a C API and a kernel programming language – OpenCL C. On AMD devices, there are two drivers that support OpenCL C – AMDGPU-Pro and ROC's HSA runtime API. [35]

In [35], the authors evaluate performance of ROCm programming frameworks – namely OpenCL, HC++, and HIP. Unfortunately, comparison with OpenMP is missing there. The authors use several real-world workloads and benchmarks to evaluate performance of the different frameworks on the same hardware. They conclude that from this trio – all running on ROCm platform – is HIP the best performing solution overall. The authors also compare performance of OpenCL with AMDGPU-Pro and ROC's HSA Runtime. The ROCm version slightly outperforms AMDGPU-Pro in applications that are not bound by memory copy. On the other hand, in other applications, ROCm version tends to perform worse. The authors conclude that the main reason for this is the API call overhead of HSA platform, along with memory transfers. What's more, the paper also presents a comparison of HIP and CUDA code running on the same NVidia hardware. There, they do not measure any noticeable overhead so they could conclude that HIP brings portability while not losing on performance when compared to CUDA. Based on their findings, HIP was chosen as a portable solution for both platforms that does not have any noticeable performance penalty on either platform, when compared to the available alternatives.

2.3 MPI

MPI (Message Passing Interface) is a standard that defines an API enabling both intra- and inter-node communication between processes. It is a de-facto standard solution for running parallel programs on HPC distributed memory systems. It is also very popular for shared memory systems. First version of MPI was released over 25 years ago and the most recent version – MPI 3.1 [12] – was released in 2015. The motivation behind

creation of MPI was the desire to create a cross-vendor API standard for shared as well as distributed memory systems which would replace vendor-specific solutions that had been used previously. Having achieved that, one could ensure portability of applications across systems by different vendors and also made programmers' work more efficient as they did not need to learn to work with new solution every time they would use a different system.

The main function of MPI is to connect the endpoints (processes) and ensure both synchronization and data exchange in form of messages. The implementation is not a part of the standard; it only defines the characteristics of the API.

An MPI program is a set of processes launched, depending on the settings and system, on one or more nodes. The processes are isolated from each other, which means that they cannot access each other's resources (RMA being an exception), however they are connected to each other and can send messages to each other. An MPI process can, but may not, be realised as an OS process of the host node. All processes run the same binary, however they usually operate over different sets of data. Such an approach is called Single Program Multiple Data (SPMD). The actual synchronization and isolation of the processes, as well as implementation of the API calls is may be different for each product yet all products should comply with the MPI standard. Therefore, an application written in compliance with MPI standard should work on all correct implementations. However, this does not work the other way around – an application that works on one MPI implementation may not work with other implementation or system.

There are multiple efforts to implement the MPI standard – both open source and proprietary. The popular ones include:

- OpenMPI
- MPICH
- MVAPICH2
- Intel-MPI

2.3.1 Communication types

It is possible to classify MPI communication types by different measures. One of them is the number of involved parties – MPI offers means to point-to-point and collective communication. Another possible classification distinguishes one- and two-sided communication depending on how many parties are actively involved in a message transfer. These could be further divided by other characteristics, such as whether an operation is blocking or non-blocking.

Point-to-point vs. Collective communication

Point-to-point communication Point-to-point communication includes only two parties. A typical example is a send-receive operation. One party, the sender, posts a send operation, and the receiver posts a matching receive operation. The operations are always matched and the operations are used as a hint for the MPI engine to enable and process the

operation. Typical API call examples are different variants of send and receive operations: `MPI_Send`, `MPI_Recv`, `MPI_Isend`, `MPI_Irecv`.

Collective communication Collective communication involves multiple parties, typically either one party spreading its data to many others (broadcast – `MPI_Bcast` or scatter – `MPI_Scatter`), or all involved parties combining its data to compute a result, for example a sum (reduction – `MPI_Reduce`). There is a plethora of collective operations that define combinations of collecting, combining, and spreading data from multiple parties.

Moreover, there is another special use-case for collective communication – synchronization. MPI offers means to synchronize among processes, meaning all specified[2] processes have to reach the barrier in order for the processes to be allowed to continue computation. Such a synchronization point is created with `MPI_Barrier`.

Collective operations are usually considered two-sided communication. As collective communication operations, apart from synchronization, are not used in the implementation of our solutions, we will not get more in depth on collective communication.

One- vs. two-sided communication

Two-sided communicaiton Two-sided communicaiton is the classical communication model where all parties – most typically the sender and the receiver – post MPI calls that trigger the data exchange. Two-sided communication covers both point-to-point communication, such as send-receive operations, and collective communication, such as broadcast. Technically, one could imagine a broadcast operation as a sequence of send-receive operations. In order for two-sided communication to work properly, several criteria have to be met. All parties need to post matching API calls for the communication to succeed. For example `MPI_Send` and `MPI_Irecv` match, while `MPI_Bcast` as sender and `MPI_Recv` as receiver do not. What's more, parameters of the operations, such as the sender and receiver ID, size of the message, tags, and the communicator, need to match as well to ensure correct routing from the source to the destination.

One-sided communicaiton Unlike two-sided communication, one-sided communication does not need a tuple of matching API calls to work. Instead, one-sided communication uses RMA (Remote Memory Access) to directly access other process's memory without having to communicate with the target CPU or involving it otherwise. Such a memory region has to be first created by the host with `MPI_Win_create` to allow RMA access to the specified memory segment. Then, the memory from that segment can be written to with `MPI_Put` or read from with `MPI_Get`. The accesses can be synchronized by `MPI_Win_fence`, which partially resembles memory fence operations.

Usually, one sided communication is point-to-point. There are also approaches using RMA in some stages of collective operations, such as in [20], however the communication itself still remains two-sided.

[2]The subset of processes involved in synchronization and other collective operations is defined by a communicator – a construct that contains a subset of processes. `MPI_COMM_WORLD` is a default communicator containing all processes.

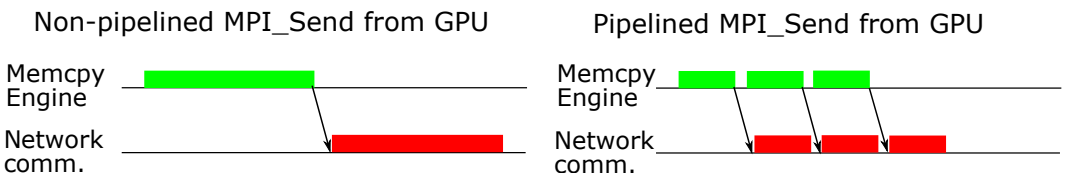


Figure 2.4: *Pipelining of memcpy from GPU memory and MPI_Send.*

Blocking vs. Non-blocking communication

Some MPI calls wait until the API call has completed, whereas others might return immediately, while the task is still being in progress. An example of a blocking MPI call can be `MPI_Recv` that blocks the stream execution until the receive buffer has been written to. Contrary to that, `MPI_Irecv` is a non-blocking call. There, the programmer defines the parameters of the received message, however the message can arrive at a later point while the host had worked on other tasks, for example. In order to still have control of the data flow, MPI offers API calls that check the status of a non-blocking call, such as `MPI_Test`.

2.4 Related research and technologies

Our work focuses, broadly speaking, on communicating data from one GPU to another one with MPI as a layer communicating in between the processes that command the GPUs. There are some existing technologies and there has been a lot of research and a lot of approaches on transmitting data between GPUs.

The majority of the technologies and research available are focussed on CUDA and therefore NVidia devices exclusively. However, it is beneficial to have an overview of those technologies and approaches as well. Some of those technologies exist with an equivalent or similar functionality for AMD hardware, others work exclusively on NVidia devices for the time being.

2.4.1 CUDA-aware MPI

CUDA-aware MPI is a broad term that describes an MPI implementation that has built-in functionality for cooperation with potential CUDA devices attached to the node. The functionality can usually be enabled and disabled.

The most common use-case for a GPU-MPI interaction is either sending data located on a GPU over to another MPI process or receiving data for a GPU over MPI. The naive algorithm would copy the data from GPU memory over to the one of the CPU and then trigger MPI communication (or vice-versa for receiving). However, thanks to UVA (Unified Virtual Addressing, introduced in CUDA 4.0), all data on the whole

system resides on a single virtual address space. That means, a subspace that belongs to main memory, another subspace that belongs to the first GPU, and possibly another subspaces that belong to the subsequent GPUs or other devices attached have unique address spaces. This way, each address clearly belongs to the main memory or one of the GPU's memories, and the physical location of each piece of data can be identified based on its address. An MPI installation aware of this can use any of the physical memories as a source or target buffer for the transmitted data. Then, the programmer does not have to specifically copy the data between CPU and GPU as this can be taken care of through the CUDA-aware implementation transparently to the programmer. Moreover, depending on the implementation, it can be optimized by pipelining those two data movement operations, as sketched in fig. 2.4. [17]

GPUDirect

The main performance gain of GPUDirect is its ability to share page-locked memory segments between GPU and MPI. A brief description of page-locked (pinned host) memory was provided in section 2.1.1. Memory copies between CPU and GPU go through main memory's page-locked segment. The same segment is used by MPI to putting data on the interconnect. Normally, a memory transfer via MPI from GPU memory goes in the following way in case of send operation: `GPU memory` → `page-locked CUDA memory (on main memory)` (→ `pageable memory`) → `page-locked MPI memory` → `network`; and reverse for receive operation. However, with GPUDirect, the copies between CUDA and MPI page-locked memory segments can be omitted, as they can share the same segment. This way, additional memory copies are omitted, therefore the performance is better because there is fewer memory transfers needed. With GPUDirect, a piece of data goes through the following locations in case of send operation: `GPU memory` → `page-locked memory shared by CUDA and MPI (on main memory)` → `network`; and again in reverse for receive. [17]

GPUDirect RDMA

GPUDirect RDMA technology is an optimization of GPUDirect. The main idea of this technology lies in MPI being able to access GPU's memory. The data can be transferred from GPU memory directly on the NIC and on the interconnect network. This approach brings two advantages: first, there is less data movement needed in order to perform an operation, and second, the main memory or CPU are not involved in the data movement as the data goes directly from the GPU on the network. This factor reduces stress on the main memory. On the data movement path for a send operation remain only `GPU memory` → `network` and the same in reverse for receive. [17]

GPUDirect Async

GPUDirect Async is an evolution of GPUDirect RDMA. The functionality aims on enabling asynchronous MPI communication from GPU kernel's perspective, so that the GPU can trigger MPI operations that were previously prepared by the CPU. This approach can help overlap kernel launch overhead with useful computation – for example, the GPU

kernels can be actively waiting for an incoming MPI data transfer instead of launching them after the operation on CPU finishes. However, the communication is still restricted to the kernel boundaries, therefore the overhead of terminating and launching new kernel still impacts the performance. [9, 18]

Research related to CUDA-aware MPI

There have been many research papers published on extending and optimizing the CUDA-aware functionality for various MPI operations and implementations. The authors in [18] compare existing approaches to network communication triggered by GPU. They point out, among other technologies, GPUDirect Async, as one of the recent promising approaches, however they still find the possibility of communication only at the kernel boundary limiting. The reason for that are kernel launch times, which become significant in modern architectures when needing to communicate over the network frequently. "Depending on the size of the kernel stream presented to the scheduler and the details of the target hardware, the launch latencies can vary from 3 µs - 20 µs" [18]. As they point out, the kernel launch overhead becomes an important issue hindering the performance, so they, as well as we, propose a way of communication that does not require killing the kernel. To overcome the problems presented by the existing approaches, they propose an approach 'GPU Triggered networking' where GPUs can post messages directly to NIC. The NIC contains structures previously created by the CPU, that can be located by matching tags. The system can communicate without GPU kernel needing to be killed. Similarly to [18], the authors of [8] propose a framework on one-sided RDMA communication that can trigger network communication from the GPU itself, so that the GPU can communicate without CPU interventions.

The authors in [40] present a GPU-aware extention to MVAPICH2, along the lines of pipelining and overlapping of asynchronous CPU-GPU memory copies and asynchronous MPI communication as sketched in fig. 2.4. [13] focuses on optimizing CUDA-aware MPI for managed memory.

In [15], the authors present an approach to make some functionalities of CUDA-aware MPI available also for OpenACC+MPI and the authors of [28] utilize AMD's DirectGMA technology – a competitor to NVidia's GPUDirect – to make low-latency high-bandwidth DMA data transfers between FPGA and GPU with OpenCL.

An analysis of the available solutions performed at LLNL showed that CUDA-aware MPI-related technologies do not perform well enough in mitigating the overhead caused by involving GPU computation. Their findings can be summarised to the following points:

- When using CUDA-aware MPI, the MPI communication is not triggered before all kernel functions have finished.
- The overhead for launching a kernel multiple times – once for packing the data to send and once for unpacking the received data – is high and should be reduced.

- The NIC buffers (on Sierra supercomputer) are not large enough to efficiently utilize GPUDirect RDMA or GPUDirect Async. However, this limitation is specific to LLNL's hardware, and may not be an issue at all on other systems.

2.4.2 CPU-to-GPU Callbacks

The authors of [33] developed an approach that lets the GPU overtake some of the CPU's usual responsibilities, such as triggering hard-drive access, network communication, or system calls. The provided implementation, however, does not aim at high performance – it rather eases up the programming effort and makes the code more future-proof. The CPU still remains responsible for performing the tasks. These are, however, requested and triggered by the GPU kernels.

The implementation uses zero-copy memory which can be accessed by both the CPU and the GPU. The GPU can therefore create requests for the CPU by writing to this memory. Equally, the CPU can write to this memory segment to let the GPU know that the callback has finished. This way, GPU kernels can run asynchronously and do not need to wait for the callback to finish. Still, the GPU kernel can poll the status of the callback.

In our approach, a simplified variant of this general idea was used. As there is only a handful of very specific tasks in our approach, there is no need for large generality, such as argument passing logic or using special callback handles, that impact the performance.

There are also other approaches that allow commanding CPU from the GPU. For example, [38] focuses on triggering CPU system calls from the GPU, however this approach is less relevant as our focus is not on system calls.

2.4.3 Other relevant research

MPI API calls from GPU There are also approaches that focus on triggering MPI communication from GPU, such as [22]. It enables triggering MPI calls directly from GPU kernel. Similarly to the other approaches, there is a callback to the CPU that performs the MPI call in the background. When waiting for the CPU to execute the MPI request, the GPU kernels have to break. The authors implement this by terminating and starting the kernels again. The registers are stored in and loaded from the GPU global memory to enable continuity of the program flow. This pattern goes against our aim to start the kernels only once per iteration.

Apart from previously mentioned research efforts, there are other research focusses worth mentioning, although they are not that closely related with our work:

Approaches on integrating task-based heterogeneous CPU-GPU programming with MPI, such as [6, 31], present ways to dynamically schedule work on host or device.

In [32], the authors propose a universal approach to efficiently pack and unpack various MPI non-contiguous data types on GPUs before or after inter-process communication with MPI. The framework contains multiple schemes that correspond to typical periodicities or shapes of non-contiguous data and selects one of them based on the data type to achieve good performance. A similar, however not as universal approach is proposed in [39]. Both of the approaches were proposed to extend MVAPICH2.

2.4.4 Learnings from related research

There are several patterns that were presented in this section and that could lead to efficiently implementing our solution. Even though CUDA-aware MPI was not used in our solutions mainly because of its lock-in to NVidia hardware, its concepts are still worth considering. Mainly the pipelining of CPU-GPU and MPI communication could bring a speed-up to our solution as well. Moreover, pipelining of those operations should not be hard to implement in HIP so that it works in a platform-agnostic manner – both on NVidia and AMD devices. On top of that, one can consider pipelining the solution at a more general scale. This will be presented more in detail in section 3.6.

Concerning CPU and GPU communication, as well as [33], our solutions also utilize zero-copy memory as main interface for communication. Even though we do not need such generality and focus rather on performance, the main idea of our approach is similar to the one presented in [33].

'GPU Triggered networking' presented in [18] is also an approach worth examining further, but this remains out of scope of this work as the implementation of that would not be very straightforward on AMD devices. Moreover, this solution could also lead to performance degradation for the same reasons GPUDirect RDMA did not work on LLNL's Sierra.

2.5 Description of the use-case

In chapter 1, there was a brief summary of the use-case being investigated in this work – the goal is to perform halo exchange between cells located on different GPU nodes. This section will provide more details that specify the use-case further.

We consider a general stencil code with halo exchange at the end (or at the beginning) of each iteration. The stencil code remains undisclosed. When optimizing the performance of the application as a whole, this lack of information would be a limitation. With such limited knowledge of the problem, it is only possible to restrict oneself to a general scenario. As the stencil code is unknown, it is for instance not possible to experiment with overlapping communication (halo exchange) with computation (stencil computation), which is a frequently used technique for increasing performance in such cases. On the other hand, being oblivious to the actual use-case enables developing a universal solution that can be applied to a plethora of implementations that fit the general scenario – distributed stencil computation on multiple GPU nodes.

Such a demarcation of the use-case sets the overall workflow of the application. The application runs for multiple iteration and each iteration consists of an arbitrary stencil computation that takes place on GPUs and a communication round. Before the communication procedure starts, the data relevant for the communication is located on the main memory (CPU). Nevertheless, the data produced by the stencil computation is not ready to be sent directly. Instead, it has to be preprocessed before sending, which consists in writing data to a contiguous *send buffer*. This procedure will be referred to as **packing**. Analogously to that, the same procedure in reverse (will be called **unpacking**) takes place on the receiving side. Unpacking is a procedure where the data received in one piece in *receive buffer* (or *recv buffer*) is processed the inverse way to packing and as a result,

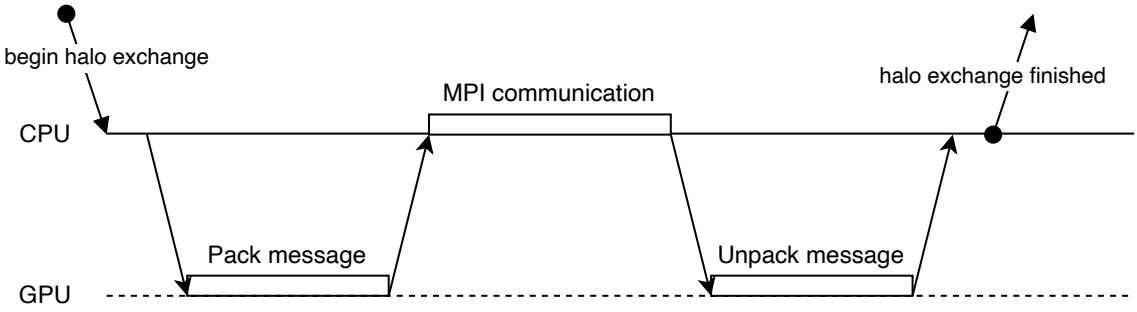


Figure 2.5: *General workflow of the halo exchange. First, the GPU packs the message to send in send buffer. Next, the CPU communicates over MPI. Once the communication finishes, the GPU can unpack the message in recv buffer.*

the data is available for the next stencil iteration in the form and on the address the next stencil iteration expects it. Our implementation remains oblivious both to the stencil code as well as the way the data is packed and unpacked. In the measurements, mock workloads were used instead. There are multiple possible use-cases for implementing packing and unpacking functionality. The most common and simplest is gathering non-contiguous data into a contiguous buffer to ease MPI communication when the MPI implementation does not deal well with non-contiguous data transfers. Other, less common use-cases, may optimize packing routing to skip sending some boundary data that do not change between iterations so the old value remains up-to-date. Next, it can be used for compressing or aggregating the data so that the amount of data sent over the network is reduced. Finally, one may need to encrypt and decrypt the data if there is a need to not send the data in plaintext over the network. Regardless of the use-case, the packing and unpacking is parallelized and takes place on the GPU as well, in order for it not to be the bottleneck that hinders the performance of the application.

The halo exchange consists of multiple steps. Before it begins, the stencil computation resulted in writing to multiple memory locations on the CPU. First, the input data needed for the packing function needs to be made available for the GPU. Then, the packing takes place. As soon as packing finishes, the resulting buffer is communicated to another GPU (be it on the same or different node). Finally, the received data is processed (unpacked) on the target GPU and made available to the CPU so that it can start the next stencil iteration. Each GPU block has its own buffer so this process is performed for each block. The workflow can be seen in fig. 2.5.

Halo exchange in general needs to deal with parallelism on multiple stages:

- There are multiple MPI processes (ranks) – be it on one or multiple physical nodes – that communicate among each other.
- Each MPI process may be split further in threads to deal with concurrent tasks on the CPU side more efficiently. We, however only consider one GPU per MPI process.
- Each GPU is split into blocks that work independently on each other. They work on

independent sets of data and the resulting data of each block may be communicated with different MPI rank.

- Each GPU block is expected to exploit GPU's SIMT and SIMD parallelism., i.e. it consists of multiple threads.

2.5.1 Detailed specification

There are some details that further specify the examined use-case and set some boundaries to the problem dimensions.

- The lifetime of kernel functions for packing and unpacking is bounded by the halo exchange segment. That means, the kernel functions cannot start before the current iteration of stencil computation has finished and it must be terminated before the next stencil iteration begins.
- The grid size for packing and unpacking is (up to) 27 blocks. The grid consists of $3 \times 3 \times 3 = 27$ blocks. Each block may consist of multiple threads.
- Even though each block may exchange data with a different GPU, an arbitrary block b always sends data to block b and receives data from block b.
- The size of the *send* and *receive buffers* may differ for each block and can range from 8 B to 1 MB ($=1 \times 10^6$ B). The sizes of the *send buffer* and the *recv buffer* are the same for each block. Obviously, block b has the same size on both the sender and the receiver GPU.
- The size of each block is defined at the beginning of the program and does not change.
- The sender and recipient of each block's buffers are defined at the beginning of the program and do not change.
- The halo exchange finishes when all blocks of all GPUs finished unpacking, the unpacked data is available to the CPU, and the unpacking kernels have terminated.

Figure 2.6 illustrates a possible communication scheme for 4 GPUs, which in our case also maps to 4 MPI processes. In this example, different blocks receive data from and send data to different GPUs. While for block 0, the data is sent to and received from $rank + 1$ for even and $rank - 1$ for odd indices, for blocks 1 and n, the send operations are directed to $(rank + 1)\%size$ and $(rank - 1)\%size$, respectively. An interesting observation is that for block 1 (and analogously for block n), all GPUs have the same buffer sizes. This is the case due to the fact that for those blocks, all GPUs are connected with each other. Contrary to that, communication concerning block 0 can be split into two subsets where the GPU communication scheme is interconnected – {GPU 0, GPU 1} and {GPU 2, GPU 3}. The buffer sizes on block 0 can therefore be different for those subsets.

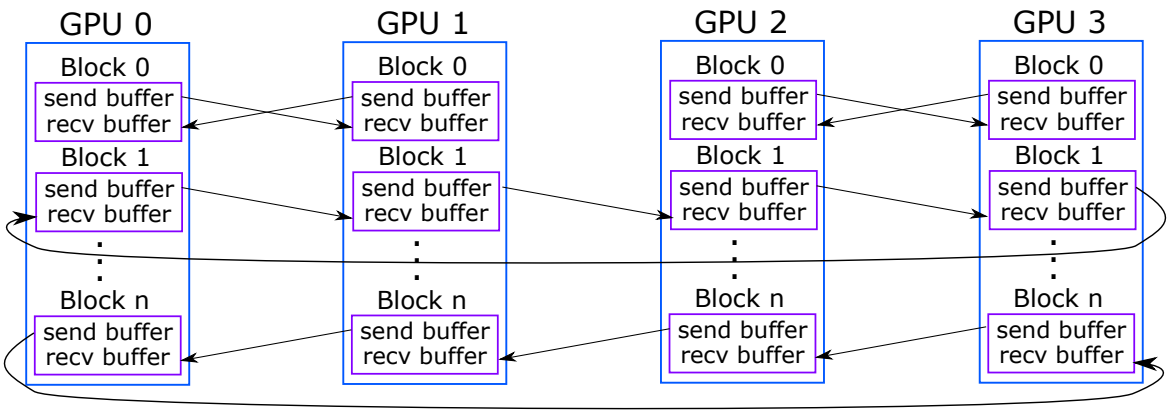


Figure 2.6: *An example communication scheme. Block x always communicates with block x of another GPU. In this example, block 0 exchanges data pairwise – with preceding and succeeding GPU, respectively, data of block 1 is sent to the succeeding GPU and data of block n is sent to preceding GPU.*

3 Design and Implementation

Transferring data between GPUs in halo exchange was examined from many perspectives. On the top of the naive baseline implementation, an alternative approach, that will be referred to as *taskqueue*, was developed. Within this general framework, multiple concepts, that can potentially increase the performance, were considered. Some of these concepts were implemented in order to prove or disprove the performance gain they bring. As these concepts develop different ideas and target different aspects of the process, it is possible to combine more of them into one implementation.

Therefore, one can assign each design decision a dimension and the resulting space will cover all possible options resulting from combinations of the concepts. The following concepts, that were considered and implemented as different variants of the *taskqueue* approach, will be discussed in this section apart from the baseline implementation:

- **Task queue** vs. **per-block channel** – The original task queue approach uses a single task queue per CPU where all GPU blocks communicate with the CPU. A variation of this is an approach where each GPU block has its own channel for communication with the CPU.
- **Pinned memory** vs. **Memcpy** – The GPU reads from and writes to pinned (zero-copy) memory in the first case, or uses memory copies between the host and the device to work with local data. The former is the 'standard taskqueue' approach. The latter will be referred to as *memcpy* or *device memory* approach.
- **MPI blocking** and **non-blocking** communication – The send operations are done either with blocking (`MPI_Send`) or non-blocking (`MPI_Isend`) MPI calls. The receive operations are always non-blocking `MPI_Recv`.

Having many possible implementations resulting from the combination of the different options, it is possible to implement them and measure the performance both against each other and against the baseline code. Tables 3.1 and 3.2 (3.1 for baseline and 3.2 for taskqueue) show the concepts that were developed and the options that rise from combining the approaches.

Baseline approach comes in 4 variants for NVidia devices and in 8 variants for AMD devices. The purpose of implementing multiple variants is that it helps assessing the performance of the taskqueue approaches. First, the same type of MPI communication can be used as a reference, regardless of using blocking or non-blocking communication in the taskqueue approach. Second, the effect of using `MPI_Waitall`, which is the default and intuitive option for baseline, can be compared to `MPI_Testany`, which complies with taskqueue approach workflow. Next dimension is presented by the 2 coherence options for pinned host memory – coherent and non-coherent. The non-coherent pinned host memory

Baseline versions

	MPI_Waitall	MPI_Testany
Using HIP coherent memory:		
MPI non-blocking sends	✓	✓
MPI blocking sends	✓	✓
Using HIP non-coherent memory:		
MPI non-blocking sends	AMD only	AMD only
MPI blocking sends	AMD only	AMD only

Table 3.1: *Matrix combining implementation options for baseline approach. There are 5 different resulting implementations for NVidia and 10 for AMD devices available.*

Taskqueue versions

	Task queue	Per-block channel
Using pinned host memory:		
MPI non-blocking sends	✓	✓
MPI blocking sends	✓	✓
Using device memory + memcpy:		
MPI non-blocking sends	✓	✓
MPI blocking sends	✓	✓

Table 3.2: *Three-dimensional space formed by combining different concepts for the taskqueue approach. As each dimension presents 2 possible implementations, there is 8 options in total.*

is a construct that was only available on our AMD testbed. The non-coherent alternative was implemented in the baseline code in order to assess the performance gains and the general capabilities this construct brings. Concerning taskqueue implementations, the previously presented concepts form a three-dimensional space, which results in 8 possible combinations, and, along with baseline approaches, will be described in detail in this section.

On the top of these different concepts, one of the targets of this work is to examine portability between vendors and therefore, as mentioned in chapter 1, our implementations were tested on AMD and NVidia GPUs (more details in section 4.1). Therefore, this distinction may present another dimension. The performance of the implementations presented in tables 3.1 and 3.2 on both platforms will be discussed in chapter 4.

This chapter contains description of approaches and concepts we developed and information about how these approaches were implemented. It is organized as follows: First, in section 3.1, general information on implementation is given. This includes explanation on how the code is organized, how the general workflow of the halo exchange is implemented, where the data was allocated, and how different approaches are compiled. Then, section 3.2 gives detailed information about the workload used for packing and unpacking and about the implementation of packing and unpacking functions for different approaches.

Next, section 3.3 explains the workflow, algorithm, and implementation decisions taken to implement the baseline code. Section 3.4 presents the taskqueue approach and section 3.5 describes the changes needed in the concept as well as in implementation to implement a variant performing memory copies. Section 3.6 presents other possible optimizations that are worth examining, but were neither implemented nor tested, and finally, section 3.7 discusses some differences between the platform we observed.

3.1 General implementation patterns

For the implementation, HIP C++ runtime API and kernel language was used. This covers both the host and kernel implementation. The communication layer is realised via MPI. The code is compiled with a *makefile* and is organized into multiple files that simplify orientation in the code and enable reusing certain functionality.

3.1.1 Code organization

In order to reliably compare performance of different approaches that were implemented, the code defining the workload, the data structures and the workflow of the algorithm is shared among all approaches. For all implementations presented in this chapter, one general implementation of packing and unpacking functions (called `GPU_Write` and `GPU_Read` for packing and unpacking, respectively) is used (see section 3.2). This way, regardless of the implementation, all approaches have the same workload for packing and unpacking, and therefore there is no risk that differences in workload of multiple implementations would degrade the quality of our measurements.

Sharing one implementation for multiple approaches was not restricted only to defining the packing and unpacking functions globally, the same pattern was used for other shared functionalities as well. A general rule of thumb for the implementation was that whenever a certain functionality was not specific to one approach only, it was implemented globally and therefore the same piece of code could be reused multiple times. This approach brings multiple advantages: First, having one definition of a certain functionality ensures the same workload is done when being reused for multiple approaches. When comparing performance of two (or more) approaches, one does not need to worry about different performance differences among multiple implementations being caused by different implementations of the same functionality. Therefore, the overall differences in performance can be attributed only to those parts of the code that differ in functionality. This behaviour is desirable as the goal is to observe performance differences when changing certain parts of the logic bounded by the actual change. Second, code without duplicate implementations of the same functionality is a lot easier to maintain, optimize, and understand. Moreover, further optimizations made to a particular functionality are immediately adopted by all approaches utilizing this functionality.

The main disadvantage of this approach is that it is not the most efficient option in terms of performance. Nevertheless, we decided to trade off a tiny decrease in performance for readability and clarity of the code and its tidiness. This decision made reasoning about performance data a lot easier. The suboptimal performance manifests itself for example through redundant calls of wrapper functions, storing redundant data or passing

unnecessary arguments to functions that are only needed by another implementations. However, as none of these events occur repeatedly within one iteration and as today's advanced compilers can optimize the code to mitigate the negative effects to a certain degree, the performance penalty stays very low.

Nevertheless, there are also exceptions where a code optimization is necessary. For example, in a parallel implementation of packing function, a value is written to the *send buffer*. The following two code snippets – 3.1 and 3.2 – are identical in functionality, however as this operation is performed repeatedly, the performance would suffer when using the former implementation, therefore the latter was used.

```
1    template <typename T> __device__ void HaloLayer<T>::GPU_Send_Buffer_Write(int
         global_blockId, int index, T msg_text) {
2            send_buffer_arr[global_blockId][index]=msg_text;
3    }
4    ... (in GPU kernel function) ...
5    HaloLayer<double> *halo;
6    for ( int i=threadId; i < buffer_size  ; i+=num_threads) {
7        halo->GPU_Send_Buffer_Write(global_blockId, i, msg_text);
8    }
9    ...
```

Listing 3.1: *Buffer write – approach with C++-style setter method.*

```
1    ... (in GPU kernel function) ...
2    HaloLayer<double> *halo;
3    for ( int i=threadId; i < buffer_size  ; i+=num_threads) {
4        halo->send_buffer_arr[global_blockId][i]=msg_text;
5    }
6    ...
```

Listing 3.2: *Buffer write – approach directly setting public class variable's value.*

Organization into classes and global functions

As the implementation reuses large parts of the code, the logical sections of the code were grouped together, so that the readability of the code can be maintained. There are several files which encapsulate certain functionality, mostly in form of C++ classes. In what follows, those files are listed with a short description of its functionality and capabilities. The paragraphs below introduce and provide a short description of the logical parts of the solution and overall list of functionalities of the implemented solutions. Selected functionalities will be described more in detail in later sections, however this list provides an overview of the project.

Kernel functions `hip_kernel_func.cpp` and `hip_kernel_func.hpp`

This tuple of files contains declarations and definitions of functions run on GPU. These are different variations of packing (`GPU_Write`) and unpacking (`GPU_Read`) functions. Moreover, function for obtaining a timestamp on a GPU (`__globaltimer`) is used during packing and unpacking.

Host functions `hip_host_func.cpp` and `hip_host_func.hpp`

A part of initialization is handled in this file so that all implementations are initialized the same way and with the same default data, such as buffer sizes or numbers of iterations or GPU blocks and threads. Apart from that, there are wrapper functions here that decide whether blocking or non-blocking communication is used. This way, all parameters to the problem size and type can be specified here.

Halo exchange data structure `hip_halo_layer.cpp` and `hip_halo_layer.hpp`

`hip_halo_layer.hpp` contains declaration of `HaloLayer` class template, which is a class template that handles data and actions concerning the `send-` and `recv_buffer` and MPI communication in general. It can allocate the buffers both on the pinned host memory and on the device, depending on the use-case. Concerning the MPI communication, it contains methods for two-sided communication, both blocking and non-blocking, using the *send-* and *recv buffers* as the source and the target, respectively. Moreover, for non-blocking MPI communication commands, `MPI_Request` objects are stored here as well. The class template enables creating instances of this class with different data types for *send-* and *recv buffers*, however *double* is the only data type that was worked with.

Task queue `hip_taskqueue.cpp` and `hip_taskqueue.hpp`

Class `TaskQueue` is responsible for callbacks between the CPU and the GPU. More specifically, using this structure, the GPU can push "tasks" for the CPU, and the CPU can notify the GPU about completion of different tasks. Moreover, this class also holds data concerning the status of asynchronous memory copies between the CPU and the GPU, which is relevant for some of the versions as well. It contains implementations of both task queue with atomic operations and callback structure without atomic

Time stamping `hip_timestamp.cpp` and `hip_timestamp.hpp`

Class `Timestamp` manages timestamping of the events both on the CPU and GPU. Apart from getting and storing timestamps, it is responsible for synchronizing CPU and GPU timestamps and processing the temporal data in general.

3.1.2 Program workflow

All approaches and implementations share the same general workflow that meets the requirements of the use-case specified in section 2.5. It can be generally split into 3 sections:

1. Initialization,
2. Halo exchange iterations (in this part, the performance is inspected), and
3. Evaluation and finalizing works.

Initialization

The main goal of the initial section is to set everything up, load all variables, initialize the data structures, and get the program ready for the measurement in general. In this section, the MPI connection is initialized, and a GPU is chosen in case there are multiple of them available to one node. In order to obtain precise data, each MPI process chooses a different GPU as long as there is enough of them. Next, variables defining the problem size are defined here:

- Number of iterations,
- number of blocks per GPU and threads per block (grid and block dimensions), and
- buffer sizes for `send-` and `recv_buffer` for each separate block.

Then, the data structures needed by the particular implementation are allocated and initialized before the measurement starts. They differ according to the implementation. Finally, an `MPI_Barrier` ensures that the next stage does not begin before all processes have finished initialization.

Halo exchange iterations

The second section is delimited by a for-loop that defines the number of iterations. In this loop, different approaches to halo exchange are implemented and also, within a single iteration, the kernels are started and terminated. Performance measurements are done only within this for-loop as it lies in the focus of the project.

Evaluation and finalizing

In the last section, the measurement data is evaluated and printed out. At the end, finalizing tasks, such as freeing allocated memory, are done.

3.1.3 Data initialization and allocation

In general, the implementations contain different kinds of data that can be categorized by several aspects. These aspects influence the way how the data is initialized, allocated, and treated in general. The flowchart in fig. 3.1 represents the decision process on where to allocate the data. One possible distinction is whether a piece of data is used and manipulated in the performance-critical section of the program – in the halo exchange iterations. If not, then it is not really important how it was implemented as it should not impact the performance. If so, another question is which party uses that particular piece of data – whether it is the CPU, the GPU, or both. A rule of thumb is that if a piece of data is used by the CPU only, it resides on CPU pageable memory, if it is used by the GPU only, it should reside on the GPU memory, and if by both, there are multiple possibilities. It would usually be placed on CPU page-locked memory, where it is directly accessible for both parties. But, in some cases, it can be allocated on GPU main memory and CPU pageable or page-locked memory.

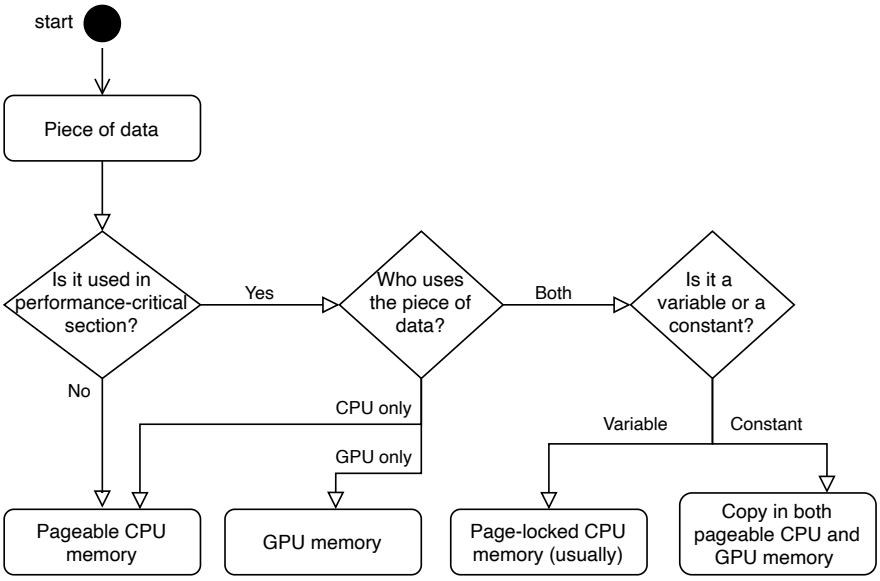


Figure 3.1: *Flowchart representing decision-making on where to store a piece of data.*

The implementation mostly follows these rules, however it is not a must. For example, classes, whose instances are needed by both CPU and GPU functions, are allocated on pinned host memory, even though some pieces of data are used only by CPUs. This was done to make the code easier to write and debug. What's more, some constant data are located only on page-locked memory. This means the GPU needs to access the main memory page-locked segment every time it needs to read it. Even though this is a suboptimal implementation and it should be changed, it does not influence the performance substantially.

As a class instance is often allocated on the pinned host memory segment, it uses an `Init` method instead of a constructor (see code snippet 3.3, the CHECK call is a preprocessor directive wrapper that checks the status of the operation). In some cases, there are multiple `init` methods available that reflect the different approaches. For example, *HaloLayer* class has init methods `Init` for general initialization, `Init_recv` for initialization including allocation of `MPI_Request` arrays for non-blocking MPI send and receive operations, and finally `Init_recv_memcpy` allocates data structures for copying the send buffer between the host and the device. Depending on the choice, the necessary data gets allocated and set up in order to provide all necessary resources for a particular approach. This approach leads to reusing the same class but not always allocating all the data if it is not needed. An alternative solution would be using C++ class inheritance but for the sake of simplicity both for the user and for the compiler, this approach was not used.

```
1    HaloLayer<double> *halo;
2    CHECK(hipHostMalloc((void**)&halo,  sizeof(HaloLayer<double>)));
3    halo->Init_recv(num_blocks, rank, size, buffer_sizes);
```

Listing 3.3: *Allocation and initialization of HaloLayer instance.*

3.1.4 Compilation

The code is compiled using `hipcc` compiler. In order to enable MPI functionality and ensure proper compilation, MPI library and header files need to be linked and included, respectively. It is done through compiler flags.

Preprocessor directives In order to prevent unnecessary branching during the runtime of the program, there are also several preprocessor directives in the code that are evaluated during compilation. Using preprocessor directives, one can compile the code for different use-cases that were mentioned in the beginning of this chapter. Moreover, there are also directives that enable other functionalities built in the code. The following directives can be used:

- `BLOCKING_SEND` directive decides whether a blocking or non-blocking (if not defined) MPI operations should be used.
- `LOCKFREE_TQ` directive decides which CPU-GPU callback mechanism is used. If not defined, a single task queue with atomic operations is used. If defined, approach where each GPU block has its own channel to communicate with the CPU is used.
- `TESTANY_AS_WAITALL` directive replaces `MPI_Waitall` operation with a sequence of `MPI_Testany` operations – if defined.
- `DETAILED_TS` defines whether timestamps of other GPU events than packing and unpacking should be taken. At the moment, it only measures the duration of appending to Taskqueue (see section 3.4).
- `AMD` or `NVIDIA` defines which device will be compiled for. Even though for HIP code, the `hipcc` compiler can translate the source code into AMD or NVidia binary code, it is not the case for inline assembly directives. As they are an essential part of the timestamping implementation, these preprocessor directives can switch between either of the implementations for the respective platform.
- `VERBOSE` and `V_VERBOSE` are two levels of printing state of the program for debugging purposes. When defined, more information is printed, however the performance is affected by that.
- `SANITY_CHECK` directive includes logic that tests the integrity of the code to make sure the implementations work properly.
- `NON_COHERENT_BUFFERS` directive decides whether to allocate buffers using coherent or non-coherent pinned host memory. If set, the buffers will be allocated using non-coherent memory.

3.2 Packing and Unpacking

Packing (`GPU_Write`) and unpacking (`GPU_Read`) functions are the main computation workload of the halo exchange procedure. The workload is consumed on the GPU. There

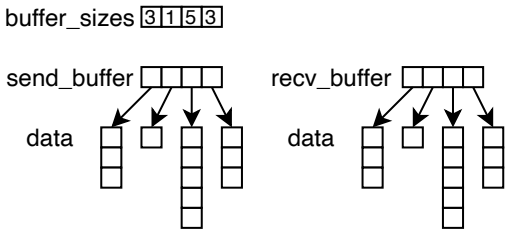


Figure 3.2: *Representation of the buffer data for HaloLayer.*

can be an arbitrary implementation of packing and unpacking functionality as long as it meets the requirements specified in section 2.5. The characteristic worth mentioning is that each GPU block is expected to work over an independent set of data. Therefore, for each block, the packing (and unpacking) functions run independently on one another.

The particular implementation of packing and unpacking functions is not relevant for the performance measurements, therefore a simple functionality was chosen for our implementation. For packing, the workload consists of filling the `send_buffer` with values. Each index of one `send_buffer` is filled with the same value, as each particular `send_buffer` is written to by one block. There is a unique value for each `send_buffer` which is based on combination of iteration number, MPI rank, and the block ID. Then, on the receiver side, one can verify from which iteration, which MPI rank, and which GPU block the value came. This serves as a proof of correct data transfer. Such procedure is shown in listing 3.4, however, this example is for sequential single-thread-per-block execution. As in each element of the array is written, the number of performed instructions is linear in the size of the buffer.

```
1     double msg_text = 1000*(iteration+1) + (rank*100) + global_blockId;
2     for (int i=0; i < buffer_size  ; i+=1) {
3         halo->send_buffer_arr[global_blockId][i]=msg_text;
4     }
```

Listing 3.4: *Unique value for each block is written to block's send buffer.*

Contrary to that, the complexity of our unpack function is lower – constant. The workload of an unpack operation used in our measurements consists of checking a single element (or a constant number of elements) of the array. The sequential single-thread-per-block implementation is shown in listing 3.5.

```
1     if(expected_value != halo->recv_buffer_arr[global_blockId][0]) {
2         assert(0);
3     }
```

Listing 3.5: *Testing the receive buffer for expected value.*

In these listings, `halo->send_buffer_arr` and `halo->recv_buffer_arr` were presented as the arrays. They both are two-dimensional arrays for *send-* and *recv_buffer*, respectively. These buffers are private for each MPI process. The first dimension separates *send-* and

recv_buffers for each block, hence the first dimension index `global_blockId`[1] points at block *global_blockId*'s buffer. The second index refers to the particular index of that array. The structure is depicted in figure 3.2.

3.2.1 Buffer allocation

From listings 3.4 and 3.5, it is obvious that both *send-* and *recv_buffer* are programatically stored in object that is pointed at by `halo`. It is a pointer to *HaloLayer* class instance. It is also obvious that both *send-* and *recv_buffer* are public as they are accessed directly. The motivation for that is shown in listings 3.1 and 3.2. Nevertheless, none of this gives an answer to where the actual data is allocated and stored.

There are two possible implementation decisions for the *send-* and *recv_buffer*:

1. Allocate the buffers on pinned host memory and access them from the CPU and GPU directly as zero-copy memory accesses. This option is used in the pinned host memory approach.

2. Allocate one copy of the buffer on CPU memory and the other one on GPU memory, and copy the data in between, when modified. This option is used in the *memcpy* approach.

Both options have their advantages and disadvantages. The advantages of the first option include using less data (no redundant buffer copies), being easier to implement (as memory consistency is transparent to the programmer), and potentially being faster as memory copies can be completely omitted. The advantage of the second option is that for GPU, locally allocated memory should be faster to access, and it can be cached by the GPU, unlike zero-copy memory which is not cached (as discussed in section 2.1.1). Summing this information up, compared to the first approach, the performance of the second approach will experience a (linear in the buffer size) performance penalty for copying the data but then, each access should be faster as lower-latency higher-bandwidth GPU memory or cache is accessed. The question is whether the lower-latency accesses can compensate for the memory copy overhead. For both packing and unpacking, memory copying takes place only twice (to and from the GPU), whereas there might be an arbitrary number of accesses to the array, depending on the implementation of packing and unpacking functions. Therefore, in theory, the first approach would be rather suitable when there are fewer accesses to the buffer, and the second one when the number of accesses to the buffer is large enough so that it compensates for the constant memory copy overhead. Also, it should be taken into consideration that all blocks of a GPU share a memory copy link so memory transfers to each block can be serialized.

Unless stated otherwise, the zero-copy memory approach is used as a basic approach. Nevertheless, in this project, both approaches were implemented. The buffer on the pinned host memory is referred to as `send_buffer_arr`, whereas the buffer on the device memory

[1]NOTE: The variable `global_blockId` refers to a global block ID within a GPU; it means that it is global in a single GPU scope, however not within the system. Each GPU has blocks numbered with `global_blockId` $= 0, 1, ..., n - 1$.

is called `send_buffer_arr_d`, if present. Implementation and concept of the memcpy approach is described in section 3.5.

Non-coherent buffer allocation

The default implementation decision is using coherent pinned host memory (see section 2.1.1 – Pinned host memory), so unless stated otherwise, the coherent option is described for simplicity. Besides, it is the only option available on NVidia devices. HIP offers non-coherent pinned host memory allocation on top of that, which is available on AMD devices.

The baseline code implements both options to compare their performance, because its workflow does not require data synchronization within kernel runtime. Non-coherent option was not implemented in taskqueue approaches, as there is no possibility to synchronize the data during the kernel runtime.

The non-coherent data works the same as coherent from programmer's perspective, one just has to bear in mind its limitations concerning coherency. Listing 3.6 presents how both coherent and non-coherent buffers are allocated. Non-coherent memory is allocated by adding `hipHostMallocNonCoherent` to pinned host memory allocation with `hipHostMalloc`. The non-coherent implementation would run on NVidia devices as well, but the memory would still be allocated as coherent in the background. It is due to the fact that the enumerators for coherent and non-coherent memory allocation – `hipHostMallocCoherent` and `hipHostMallocNonCoherent` – are represented by the same numerical value when compiling for NVidia platform, and are therefore identical.

```
1  #ifdef NON_COHERENT_BUFFERS
2      CHECK(hipHostMalloc((void**)&send_buffer_arr, sizeof(T*)*num_blocks,
           hipHostMallocNonCoherent));
3      CHECK(hipHostMalloc((void**)&recv_buffer_arr, sizeof(T*)*num_blocks,
           hipHostMallocNonCoherent));
4  #else
5      CHECK(hipHostMalloc((void**)&send_buffer_arr, sizeof(T*)*num_blocks,
           hipHostMallocCoherent));
6      CHECK(hipHostMalloc((void**)&recv_buffer_arr, sizeof(T*)*num_blocks,
           hipHostMallocCoherent));
7  #endif
```

Listing 3.6: *Coherent and non-coherent buffer allocation.*

3.2.2 Parallelization of packing and unpacking functions

Each GPU block works independently when executing packing and unpacking operations. The code snippets 3.4 and 3.5 shown previously assume sequential execution in each block – using only a single thread. However, in order to exploit GPU parallelism, the functionality deals with multiple threads per block as well – thus exploiting the SIMT parallelism on GPUs. There can be up to 64 threads in one AMD SIMT unit (wavefront), or up to 32 threads in NVidia SIMT units (warp). In total, there may be up to 1024 threads in one block. Listings 3.7 and 3.8 show the parallelized adaptation of listings 3.4 and 3.5.

In the case of the packing function, `num_threads` indices of the array are written to in parallel as the code is executed by each thread. Therefore the theoretical speedup is

num_threads for the packing function. The theoretical speedup assumes, however, that other components of the system, such as the system bus that performs the writes to pinned host memory, manage the increased workload and would not be the bottlenecks.

In the case of unpacking function, no speedup is presented. As the complexity of the unpacking functionality is constant, it is not possible to present additional speedup through vectorization. Instead, the parallelism is used to verify correctness in *num_threads* indices compared to a single one for sequential execution. From this perspective, it can be concluded that the packing function is strongly scalable whereas unpacking is weakly scalable.

```
1    double msg_text = 1000*(iteration+1) + (rank*100) + global_blockId;
2    for (int i=threadId; i < buffer_size  ; i+=num_threads) {
3        halo->send_buffer_arr[global_blockId][i]=msg_text;
4    }
```

Listing 3.7: *Parallel writes to send buffer.*

```
1    if(expected_value != halo->recv_buffer_arr[global_blockId][buffer_size/
         num_threads*threadId]) {
2        assert(0);
3    }
```

Listing 3.8: *Testing multiple indices of receive buffer in parallel.*

The number of parallel threads processing the packing function may also be used to simulate different workload on the packing function. This way, the computation-to-communication ratio can be influenced. If only a few threads per block are used, the packing function will take longer time to finish, while the size of buffers, and thus the time needed to perform data transfers over MPI, does not change.

3.2.3 `__device__` functions and `__global__` wrappers

Both the packing and unpacking functions are to be executed on the device. Depending on the approach, the functions are called either directly from the host or as a subroutine of an already running kernel function. The former must call a function defined with a `__global__` prefix, while the latter would call a `__device__` function. The difference between those two types of kernel function definitions is that for `__global__` functions, a call from the CPU triggers the whole scheduling and kernel-launching mechanism, as described in section 2.1.2. In contrast to that, `__device__` function calls from the GPU are the GPU's equivalent of CPU function call – the program execution continues using the same resources on the GPU.

In our different approaches, it is sometimes necessary to call a `__global__` packing (or unpacking) function and sometimes a `__device__` one. In order to maintain a single implementation of the same functionality, there are `__global__` wrappers for the functions in case it is called directly from the CPU, and new kernel has to be launched. The wrapper function then simply calls the `__device__` function where the implementation remains. Listing 3.9 shows such wrapper function that, after taking a timestamp, calls the actual implementation on line 3.

```
__global__ void GPU_global_Write(int rank, HaloLayer<double> *halo, int iteration,
    uint64_t *GPUts, int *GPUts_index, int GPU_total_ts_per_block){
    ... //timestamp
    GPU_Write(rank, halo, iteration);
    ... //timestamp
}
__device__ void GPU_Write(int rank, HaloLayer<double> *halo, int iteration){
    ...
}
```

Listing 3.9: *Global wrapper for device function.*

3.2.4 Device memory implementation

An additional implementation of packing functionality is `GPU_Write_device_mem`, which deals with "memcpy" implementation using two buffers (see section 3.2.1). Again, this implementation is very similar to the standard one. The only difference is that a GPU memory-allocated buffer (`send_buffer_arr_d`) is written to, in contrast to the pinned host memory-allocated one (cf. `send_buffer_arr` from listing 3.4), as shown in listing 3.10.

```
    double msg_text = 1000*(iteration+1) + (rank*100) + global_blockId;
    for (int i=threadId; i < buffer_size  ; i+=num_threads) {
        halo->send_buffer_arr_d[global_blockId][i]=msg_text;
    }
```

Listing 3.10: *Parallel writes to send buffer – "memcpy" version.*

3.3 Baseline

This section describes the baseline implementation. This is the naive implementation to solve the problem. The baseline implementation adapts some of the concepts from the taskqueue approach. It can be compiled in multiple versions that cover the combination of the approaches from table 3.1.

- It is possible to run baseline code either with **MPI blocking** or **non-blocking** communication.
- Pending MPI requests can be checked and waited for either with `MPI_Waitall` operation or with a sequence of `MPI_Testany` operations.
- (Only on AMD) The buffers can be allocated as coherent or non-coherent (see section 3.2.1 – Non-coherent buffer allocation). Coherent buffers are the default option, so they are automatically used unless specified otherwise.

Therefore, in total, there are 8 (4 on NVidia) variants of baseline. Concerning MPI blocking and non-blocking communication, the purpose of implementing both options was to have a good comparison for the taskqueue approach. As the taskqueue approach implements both blocking and non-blocking communication, the baseline has a counterpart

for both options. The implementation using `MPI_Waitall` was the original approach. The motivation for replacing it with a sequence of `MPI_Testany` operations is first to have a comparison with the taskqueue approach, and second to discover more about the status of the particular message transfers. Coherent pinned host memory is a construct known from CUDA and enables fine-grained CPU-GPU synchronization during kernel runtime. Non-coherent memory option was implemented in order to observe the performance differences between the two. As no synchronization during the kernel runtime is necessary in the baseline, the option is applicable to the baseline code.

Even though it is described as the naive implementation, many optimizations were done to the baseline code so that the eventual performance gain of the taskqueue approach cannot be attributed to simple code optimizations techniques that have little to do with our focus. It is naive in that it performs packing, MPI communication, and unpacking sequentially. This implementation is used as starting and reference point for other implementations. Identifying and mitigating the bottlenecks of this approach leads to other concepts that are described in later sections of this chapter.

3.3.1 Workflow

The baseline presents a sequential approach that exploits overlapping of computation (=packing and unpacking) and communication only to a very limited scale. The workflow of one iteration of baseline code is captured in fig. 3.3. In the first phase, packing is done and the algorithm halts until it is finished. Then, the communication section starts, and again, the algorithm waits until all messages have been received on the particular MPI process before starting the last part – unpacking. Once all threads are done unpacking and all messages have been delivered, the iteration finishes. In the next paragraphs, each of the phases will be described more in detail.

Packing

In the first phase, the CPU launches `GPU_Write` (packing) kernels for the desired grid and block dimensions. Here, the kernel launch overhead has to be taken into account. However, there must be at least one kernel launch in every iteration. The kernel performs the packing operation as described in section 3.2. As this job runs on the GPU, the CPU is free to do other work after the kernel launch call has returned.[2] The work that can be done is posting `MPI_Irecv` operations. This will prepare the MPI process for the incoming communication in the communication round.

As for posting `MPI_Irecv` operations, an API call has to be made for each block's *receive buffer*. Each of those buffers may come from a different rank; the source is known at the launch-time. Therefore, each receive operation must have the source and buffer size specified. Both may differ among blocks, but are known at the launch-time. Next, if multiple messages come from the same source, they have to be identifiable. For that reason, each receive request contains a tag – *block ID*. It is known that block b receives data from

[2]The kernel launch call is asynchronous from host's perspective. That means that the function returns without waiting for the kernel to finish.

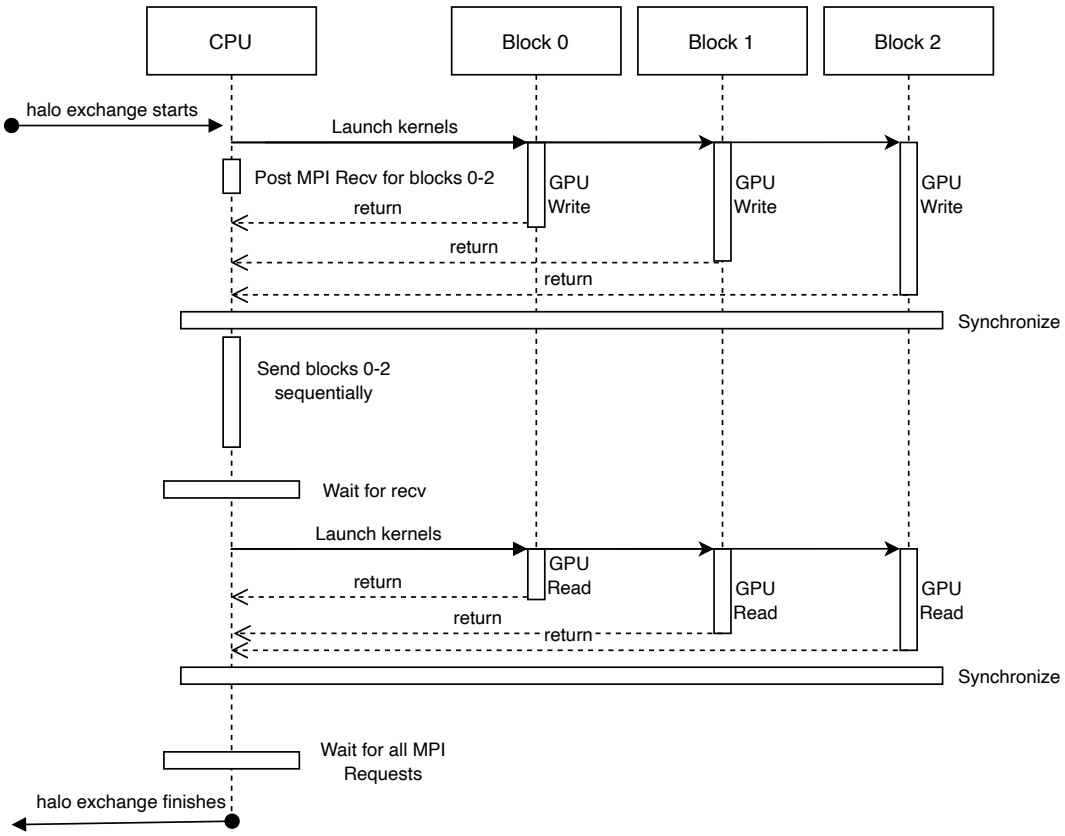


Figure 3.3: *Workflow of one baseline iteration. The halo exchange starts with launching GPU kernels for packing. The CPU waits for all kernels to finish this operation, and after that all communication takes place. When that is done, the unpacking kernels are launched and finally, the iteration can finish after all tasks were completed.*

block b on another GPU. The tag matching uses *block ID* to identify matching sends and receives. In order to enable tracking the status of the receive operation, *HaloLayer* contains an array of `MPI_Request` objects (allocated during initialization).

The packing stage on the CPU finishes with waiting for the packing kernels to finish. This is achieved with `hipDeviceSynchronize()` barrier.

MPI communication

This stage is relatively straightforward; its goal is to issue the send operations and halt the execution until data for unpacking arrives to *receive buffers*. It cannot begin before the data in the *send buffer* is available on the CPU and ready to send. As the GPU uses pinned host memory, no data movements have to be done and the outputs from the GPU are instantly accessible on the CPU. Hence, the synchronization barrier is at the end of packing phase.

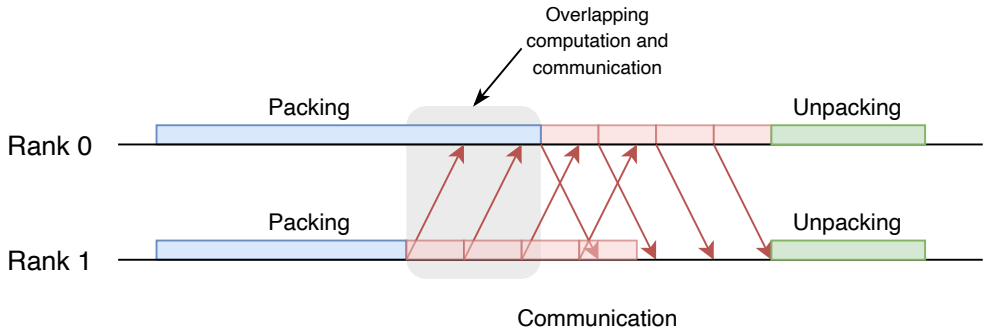
As packing may take different amount of time for each MPI process, it is possible that another process (that sends data to the current one) finished packing earlier and starts sending its buffers over. If that is the case, receiving data can take place already while packing is still in progress, as illustrated in figures 3.4a, 3.4b. That is the reason for posting the receive operations already in the previous phase. As non-blocking `MPI_Irecv` is used in all implementations[3], the CPU does not wait for the data to arrive. However, since there is no guarantee that communication and computation overlap, scenario illustrated in figure 3.4c may come to pass as well. There, the phases are strictly divided for both MPI processes.

The communication phase begins with an MPI Send operation for each block's *send buffer*. Thus, for b buffers, b send operations are posted. Depending on the preprocessor directive setting during compilation, either blocking or non-blocking sends are issued. These are `MPI_Send` and `MPI_Isend`, respectively.

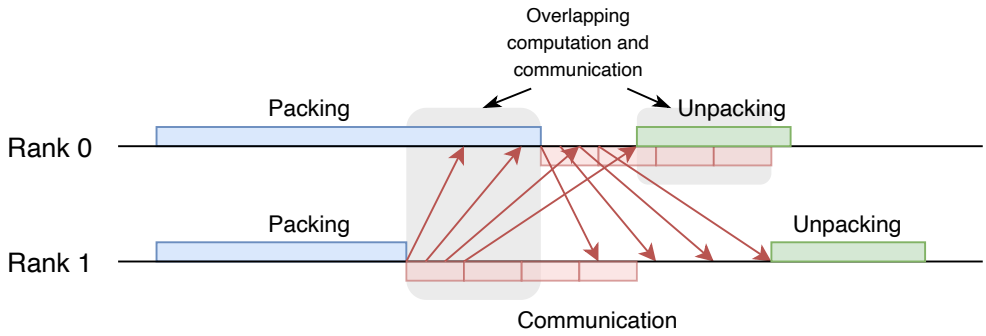
Blocking sends The blocking send operation for each block does not return until the send operation is finished. This gives a rather precise information on when each of the send operations begins and ends. Using this approach, one can ensure that only one operation per MPI rank is posted at a time. Having control over the number of operations on the network might be advantageous for systems and MPI implementations where many pending operations cause congestion of the communication channel and degrade the communication bandwidth or latency. The traffic on the network can be influenced only by the send operations, therefore posting blocking receive operations would presumably not bring any advantage in this regard.

Non-blocking sends The non-blocking sends return almost immediately and therefore, they do not give a precise information on when the operation takes place. Querying the status of the operations has to be handled with `MPI_Request` objects. Nevertheless, for this phase of the algorithm, the status of send operations is irrelevant. The advantage

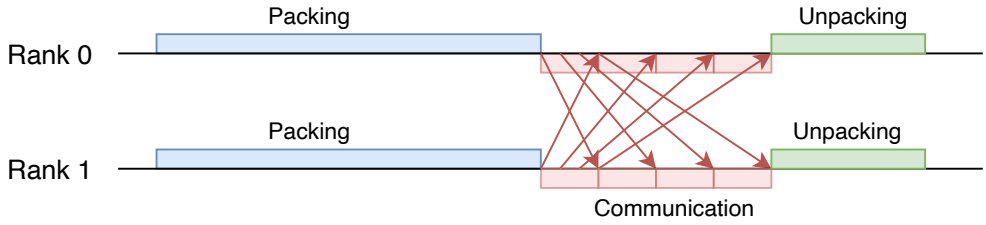
[3] Both the blocking and the non-blocking variants use non-blocking `MPI_Irecv` operation. The distinction between the two variants lies in using blocking `MPI_Send` or non-blocking `MPI_Isend` operations.



(a) *Baseline workflow with overlapping blocking communication and computation. Rank 1's packing finishes sooner and therefore rank 0 experiences overlapping of communication (non-blocking receive operations) and computation. However, then, even if all recv buffers are available, it unpacking must wait for blocking send operations to return.*



(b) *Baseline workflow with overlapping non-blocking communication and computation. Rank 1's packing finishes sooner and therefore rank 0 experiences overlapping of communication (non-blocking receive operations) and computation. As non-blocking sends are posted, rank 0 can begin unpacking while its send operations are still on the way. Nontheless, rank 1's unpacking must wait for the data to arrive.*



(c) *Baseline workflow without overlapping communication and computation. Packing segment finishes approximately at the same time, therefore no overlapping regardless of blocking or non-blocking communication.*

Figure 3.4: *Example workflows of baseline implementation that do and do not lead to overlapping computation and communication.*

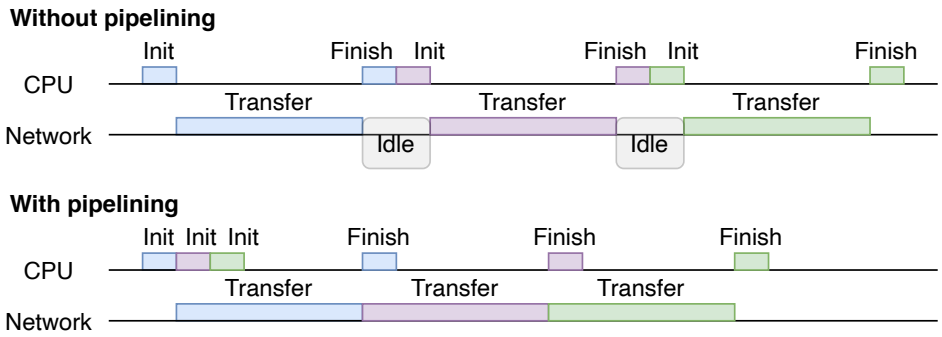


Figure 3.5: *Comparison of sequential and pipelining approaches to network communication. Pipelining enables better utilization of the network, which is the critical resource.*

of posting non-blocking send operation is that while the send operations are in progress, the CPU is not blocked. Therefore it can process other workload, such as moving on to unpacking as illustrated in figure 3.4b. Another possible advantage is that, in case the particular MPI implementation supports it, pipelining within the send operation can speed the process up if more send operations are pipelined. As the network line is likely to be the bottleneck here, pipelining can enable using (almost) the whole bandwidth. Figure 3.5 shows the theoretical gain through mitigating the idle intervals on the network. The *init* and *finish* intervals may represent, for example, data movements from and to MPI internal page-locked buffers that are used for sending and receiving the data. Pipelining of non-blocking operations is, however, not a part of MPI standard, just a possible optimization that may or may not be available.

Once the send operations are posted, the program needs to halt the execution until the data for unpacking is available; this is achieved with MPI Wait operation over previously posted `MPI_Irecv` operations. As there is an array of `MPI_Request` objects handling all receive operations available, an `MPI_Waitall` operation over this array carries the functionality out. Once the operation returns, all *receive buffers* contain the received data.

Implementation with `MPI_Testany` As the names suggest, `MPI_Waitall` blocks (waits) until all operations are finished, and `MPI_Testany` tests if any of the operations has finished. The functionality of `MPI_Waitall` can be substituted by using a sequence of `MPI_Testany` operations. The following code snippet shows how such a functionality can be implemented:

```
1 for(int i= 0; i<num_blocks; ++i ){
2     int blockId, success;
3     while(true) {
4         MPI_Testany(num_blocks, request_array, &blockId, &success,
              MPI_STATUS_IGNORE);
5         if(success && blockId != MPI_UNDEFINED) {
6             break;
7         }
```

```
8       }
9       // request for block with number blockId finished
10      // get timestamp here?
11  }
```

Listing 3.11: *Implementation of `MPI_Waitall` functionality using `MPI_Testany`.*

The end of each loop iteration reveals two potentially valuable pieces of information. First, the `blockId` variable is filled with ID of the buffer that was received. Second, if a timestamp is taken at that place, one has the information when block `blockId` was received. None of this influences the workflow of baseline itself, however it presents an interesting insight in comparison to `MPI_Waitall` where this is not possible.

Unpacking

Again, as the algorithm works with pinned host memory, no data movements between CPU and GPU are needed. So, unpacking kernels (`GPU_Write`) can be launched as soon as the communication phase is over. The previous stage only waited for receive operations to finish, so there still may be some send operations in process during this phase as the scenario in figure 3.4b suggests.

The GPUs take on the unpacking workload. Before the CPU finishes the iteration, it needs to place a set of barriers on multiple parallelism levels that assure that the iteration is complete:

1. The first barrier is needed only when **non-blocking sends** were issued. It halts until all send operations of the particular MPI process have finished. As there is an array of `MPI_Request` objects handling all send operations, again an `MPI_Waitall` can be used to deliver the functionality.

2. The second barrier – `hipDeviceSynchronize()` – blocks the execution until the unpacking kernels have finished.

3. Finally, an `MPI_Barrier` ensures that all MPI processes have reached the end of the iteration.

The set of barriers issued in this order ensures that, as the halo exchange ends,

1. all MPI communication has finished,

2. all *receive buffers* were unpacked, and

3. all MPI processes finished the iteration.

3.3.2 Non-coherent buffers

While on our NVidia testbed, only coherent option was available, the AMD testbed offers both options. The baseline approach, as presented, strictly delimits CPU's and GPU's work with the buffers. In each iteration, the CPU does not access the *send buffer* until the packing kernel terminates. Analogously, the CPU does not access the *recv buffer* after the

unpacking kernel is launched. Therefore, no synchronization is needed while the kernel is running – only at its boundaries. Thus, non-coherent memory can be utilized in the baseline code. It was not implemented in the taskqueue approach, because synchronization is necessary there.

Apart from differences in memory allocation (of `send_buffer_arr`, `recv_buffer_arr`, and the sub-arrays for each block), as desribed in section 3.2.1 'Non-coherent buffer allocation', there are no other differences in the workflow or code.

3.4 Taskqueue approach

The taskqueue is an approach that enables callbacks from the GPU to the CPU. It was designed as a response to the flaws of the baseline workflow. This section will refer to the approach using pined-host memory buffers and zero-copy accesses from the GPU. The logic of the memory copy-based variant is discussed in section 3.5.

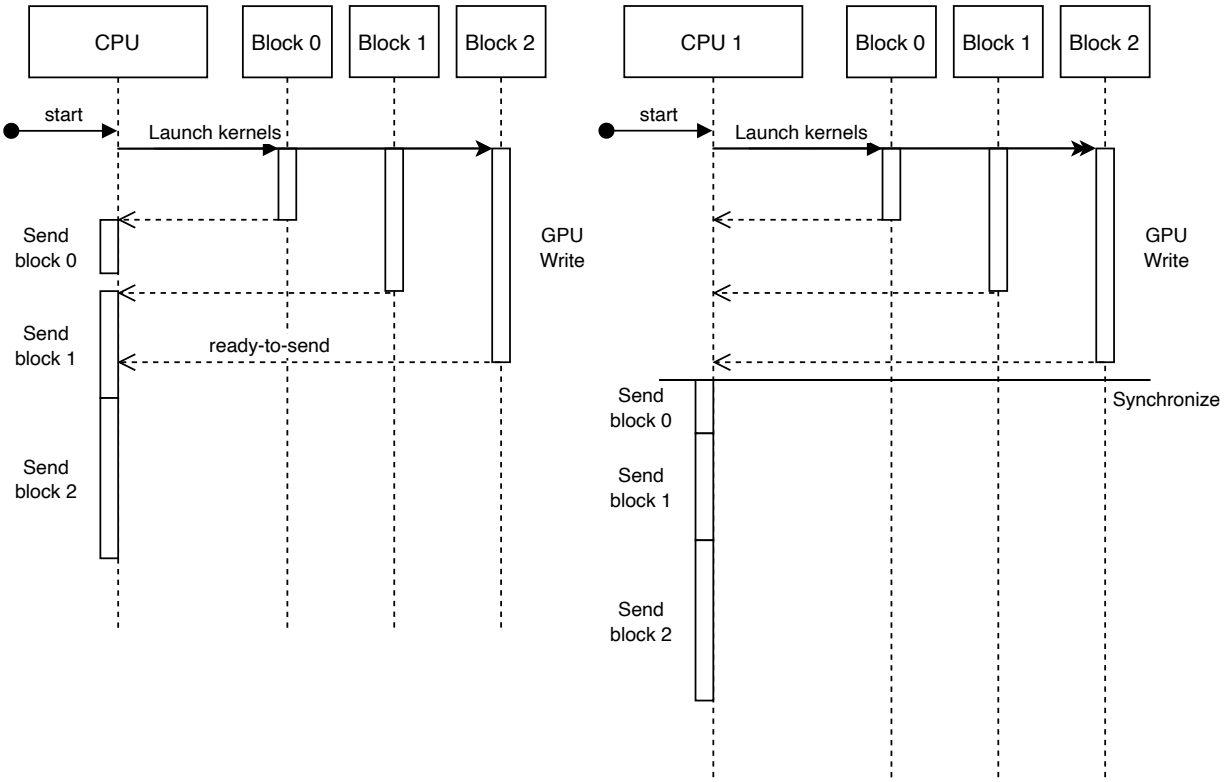
Two problematic elements could be identified in the baseline workflow as presented in fig. 3.3. First, there are 2 kernel launches – one for packing and one for unpacking – in each iteration. This means that the overhead for launching the kernels will appear twice. Second, the approach is still very sequential in terms of computation and communication handling. This means that, for most of the time, only one of the two processes takes place.

3.4.1 Goals

The goal of the taskqueue approach is, of course, to eliminate the bottlenecks of the baseline approach and hopefully achieve a performance gain through this elimination. In order to eliminate the first overhead, a single kernel function has to be designed that manages both packing and unpacking in the right moment. The second objective is more complicated to solve to a full extent. Its aim is to overlap computation and communication as much as possible.

There are two workflows in the program – the first one, where the data gets processed, which is represented by the packing and unpacking functions on the GPU, and the sedond one that represents the communication flow. The former is parallelized in two stages. First, there are b blocks working in parallel on one GPU to pack the data to b different *send buffers* and unpack them from b different *recv buffers*. Second, parallelization is present within each block as well – each block consists of multiple threads. On the other hand, communication between processes is rather sequential. The connection between two MPI processes, be it a bus or a network connection, has a given bandwidth. This means that the more data is sent, the more time will be required. In this sense, it can be said that multiple messages are being transferred in serial fashion. Hence, the sooner the communication starts, the sooner it can finish.

A possible solution to overlapping communication and computation is to trigger the communication for each block as soon as possible – while other blocks may still be computing (packing). The comparison of the concepts of the taskqueue and baseline approaches is to be seen in figure 3.6. In part 3.6a, the send operations are triggered immediately. Therefore, when the last `GPU_Write` finishes, a significant part of the communication has already taken place. This represents the aim of taskqueue approach.



(a) *Approach implemented in taskqueue.* **(b)** *Approach implemented in baseline.*

Figure 3.6: *A comparison of how and when taskqueue and baseline issue send operations. In taskqueue, a send operation for a buffer is issued as soon as the packing on that particular buffer has finished. In baseline, the CPU waits with issuing communication until all packing operations have finished.*

Contrary to this, in part 3.6b, the communication begins when the last `GPU_Write` finishes. As the communication is sequential and takes the same time to finish in both cases, example 3.6a would presumably finish the data transfers earlier. Thus, the receiving party would be able to start unpacking sooner.

Analogously, on the receiver side, the aim is to start unpacking for each block as soon as the data for the particular block is available – while other data transfers may still be pending. It will likely often be the case that the longer the packing function takes, the longer the unpacking takes. If that is true, starting unpacking immediately would not bring any advantage because the last received block would also be the one that needs the longest time for unpacking. However, it is not a rule that the packing is directly proportional to unpacking and in that case, starting unpacking early might bring additional speedup.

As the programs works with buffers of different sizes, we assume that packing and unpacking will take different amount of time for each block. The taskqueue approach profits mainly from this assumption. Nevertheless, if all blocks take the same or very similar time to finish, the computation and communication overlap would be minimal and the workflow of such scenario would look very similar to the baseline workflow presented in fig. 3.6b.

The solution for such a case was not implemented. However, a concept using pipelining was developed and will be briefly presented in section 3.6.2.

3.4.2 Concept

In order to achieve the two goals – having one kernel function for the whole iteration and triggering the communication and unpacking for each block immediately – the following concept was created. The extension of the functionality can be summed up to 2 situations that need to be covered:

1. Once packing has finished, the GPU block must trigger MPI communication for its *send buffer* and stall, and

2. once the communication regarding one block has finished (i.e. the data has arrived to one of the *recv buffers*), the particular GPU block can continue execution and perform unpacking.

The CPU remains in charge of MPI communication (cf. section 2.4). Therefore, the main idea of this approach is to enable communication between the CPU and the GPU that occurs during the lifetime of the kernel, not at its boundaries. The approach presented in [33] discussed in section 2.4.2 proposes a method in which a GPU commands the CPU to issue tasks and the CPU informs the GPU about the completion of the tasks. The taskqueue approach uses a similar concept, however it is specialised to the particular use-case of triggering MPI communication and informing of its status.

The workflow of each of the GPU blocks is modelled as a finite-state machine (FSM). Figure 3.7 shows this FSM. The initial state is entered upon launching the kernel and the final state results in kernel termination. In each state, the GPU block must complete the specified task and then progresses to the next state. CPU's workflow is not defined deterministically. Instead, it serves as a slave to the GPU and reacts to the environment –

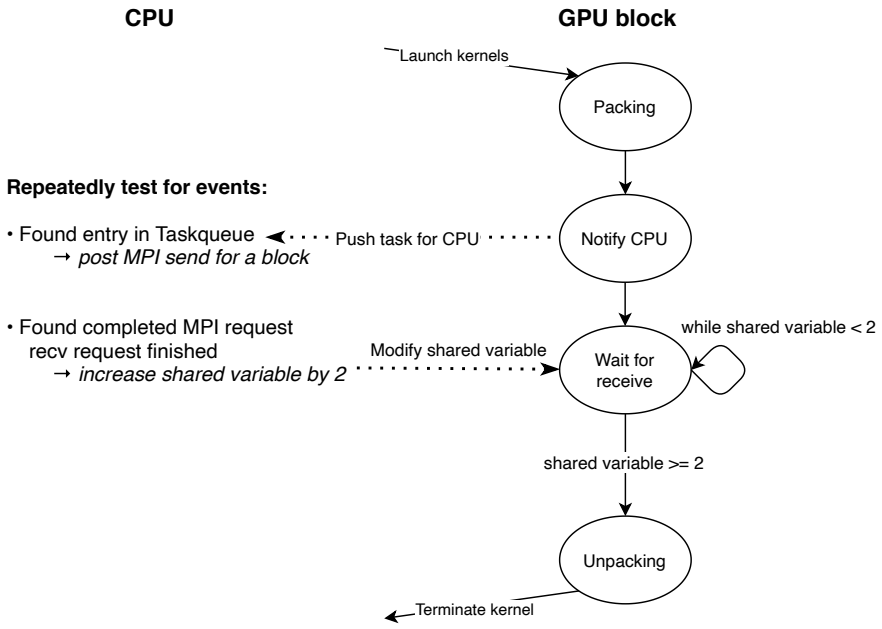


Figure 3.7: *FSM of one halo exchange iteration and one GPU block.*

that includes processing the callbacks from the GPU and reacting to completion of MPI message transfers.

The algorithm relies on the CPU and GPU being able to communicate. That is done by modifying data structures located on the pinned host memory, thus accessible to both parties. The functionality and management of the callbacks as a whole is implemented in class *Taskqueue* and kept in an instance of that class – it contains methods used both by the CPU and the GPU.

3.4.3 Workflow

Similarly to the baseline approach, each iteration begins with kernel launch and posting `MPI_Irecv` operations for each block's *receive buffer*. Once this is done, the CPU waits for events (described in section CPU waits for events below).

GPU creates tasks

The workflow of the GPU follows the FSM from figure 3.7 for each GPU block. Packing can be started immediately after the kernel launch because, again, this implementation allocates both *send-* and *recv buffers* on pinned host memory, so the data is already available.

Unlike baseline approach, there no implicit synchronization barrier after packing, so data coherency on the host is not assured. Instead, here has to be an explicit memory

barrier there. Therefore, each thread of the block calls a global memory fence operation `__threadfence_system()`, which guarantees that all write operations made by the thread before this barrier, i.e. the packing writes, will be seen by others before the write operations done after this synchronization point. Then, to assure that all threads performed the memory fence operation, `__syncthreads()` barrier is inserted.

After the packing writes are "flushed", the next step is the callback to the CPU that notifies the CPU about which block finished packing, i.e. the data in its *send buffer* can be sent via MPI. In order for this to happen, the GPU has to assign the CPU a "task". Two different concepts that accomplish this were developed – one that uses atomic operations on the GPU side and a task queue, and other one that does not need atomic operations as there is a per-block channel for communicating with the CPU. Both approaches are described in sections Single task queue and Per-block channel, respectively. Upon creating a task for the CPU, regardless of the approach, the GPU block waits for a notification from the CPU.

CPU waits for events

After kernel launch and posting `MPI_Irecv`, the CPU waits for events. The number of events is known, and is constant in each iteration. For each block, there is one event (task) from the GPU, and one event regarding completion of the MPI receive operation for the particular block, therefore $2 \times num\_blocks$ events in total. Hence, a for-loop over $2 \times num\_blocks$ events.

The workflow of processing events is presented in figure 3.8. The CPU repeatedly checks for an event until it finds one – first, it checks for a task from the CPU. If there is one, it means a GPU block has finished packing. Therefore, the host can issue the MPI send operation for the particular *send buffer*, and starts checking for the next event.

Just as in the baseline approach (described in sections Blocking sends and Non-blocking sends), there are two options for issuing the send operation – either blocking using `MPI_Send` or non-blocking using `MPI_Isend`. However, this time, only one send operation for one buffer is issued at a time. If a non-blocking send was issued, its request handle is stored and its completion will be checked before finishing the iteration analogously to the baseline approach.

If no task from the GPU is found, the CPU moves on to checking the status of the previously posted `MPI_Irecv` operations through their `MPI_Request` handles. With the array of handles, `MPI_Testany` can test if any (and which) receive operation has finished. If there is one, the particular GPU block can be notified that it can start unpacking (the implementation of notification functionality is described in section Request completed notifications). This is a similar strategy to using `MPI_Waitall` in the baseline code, however `MPI_Testany` does not block, so, if no message was received, the CPU can move on to checking the taskqueue again.

GPU can continue with unpacking

The GPU block was halted until notified from the CPU, which took place when the *receive buffer* arrived. When the notification is received, the GPU moves on to unpacking as

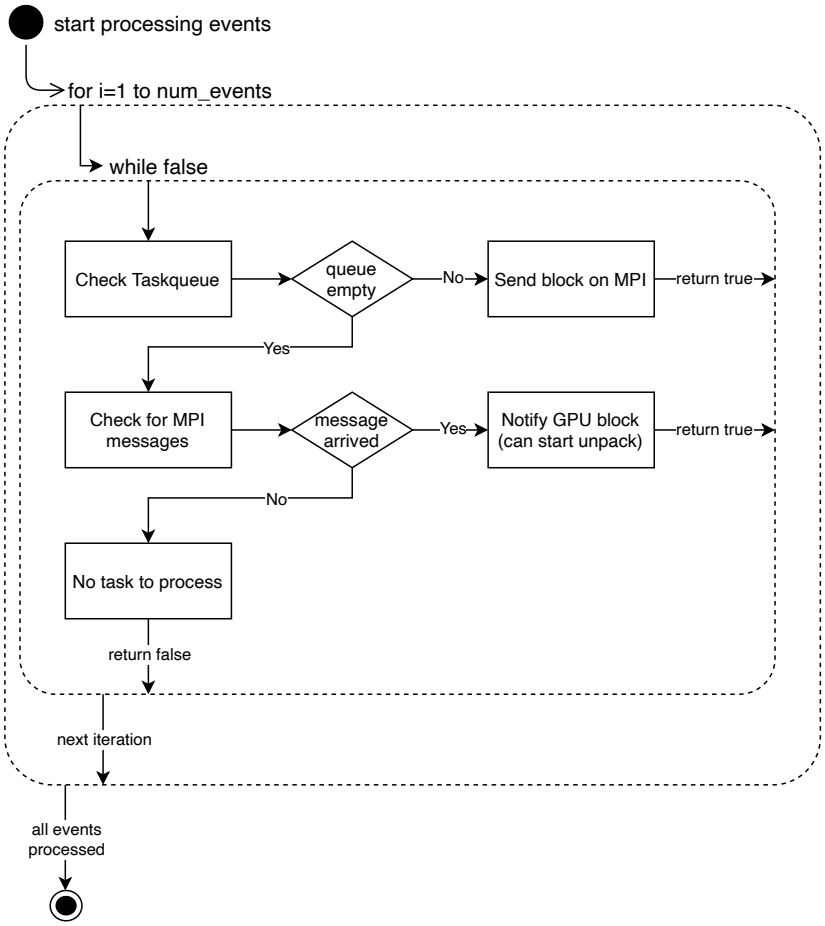
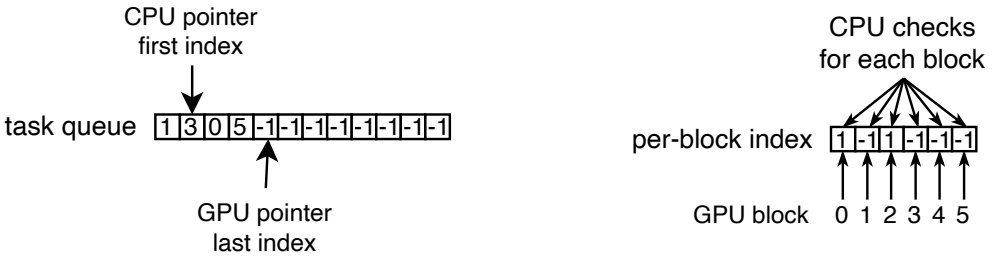


Figure 3.8: *Workflow of a CPU processing events in the Taskqueue approach.*



(a) *Single task queue approach. GPU blocks enqueue at the bottom and CPU dequeues from the front.*

(b) *Per-block channel approach. Each block has its own index (channel) where it pushes tasks fro the CPU. CPU needs to check all indexes.*

Figure 3.9: *Two possible implementations of CPU-GPU callback structure.*

depicted in figure 3.7. When the unpacking operation is fihished, the GPU block has finished its task for this iteration and can terminate.

Finishing the halo exchange iteration

As soon as the CPU processes all $2 \times num\_blocks$ events, its work in terms of the halo exchange for this particular iteration is done. Similarly to baseline approach, there are several synchronization points that the CPU has to pass. The first one is waiting for all blocks to finish unpacking (`hipDeviceSynchronize()`). In per-block channel implementation, after this barrier is passed, the CPU resets the per-block channel array (`cpuTaskList`) to initial state for the next iteration. The second synchronization point applies only to non-blocking implementation – waiting until all send operations have finished. And finally an `MPI_Barrier` that ensures that all blocks have finished the iteration.

3.4.4 Implementation details

Single task queue

Figure 3.9a shows the task queue data structure. There is a single task queue there, implemented as an array because the number of tasks is known at launch time. Hence, the CPU, in order to check whether there is a task, checks at the front of the queue. If there is a task there, it processes it and moves the pointer to the next index. If the CPU reads -1, which is the initial value, it means no task is available.

On the device side, when the GPU blocks finish packing, they push tasks to the end of the queue. Each block pushes the task to where the last index currently points at, and moves the pointer to the next index so that the next GPU block does not overwrite the last task. There is no consistency issue with new tasks propagating tardily to the CPU as long as they eventually become visible – the only disadvantage of later propagation is that the CPU can start processing it later.

Race condition elimination There is a single execution stream from the CPU side that processes the tasks. Therefore, there is no risk of race conditions on this side of the queue. Also, as the CPU only reads from the queue and the GPU only writes to it, the CPU cannot cause any data inconsistency. On the GPU side, there are, however, multiple blocks that may attempt to write into the back of the queue. In order to prevent race conditions where multiple GPU blocks push to the task queue, a locking mechanism using atomic operations is introduced. The locking mechanism is accessed and used by a single GPU exclusively, therefore all data regarding the locking and the GPU pointer to the end of the queue may be located in GPU memory. Also, should atomic operations on this set of data be performed, the operation needs to be atomic only with regard to the GPU. The inserting in the queue with the locking mechanism is shown on listing 3.12.

```
1    if(threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0)
2    {
3        int global_blockId = blockIdx.x + (blockIdx.y * gridDim.x) + (blockIdx.z *
             gridDim.x * gridDim.y);
4        while(atomicCAS(cpuTaskList_lock_d, -1, global_blockId) != global_blockId)
             {};  //block here until lock is obtained
5        volatile int index = *lastIndex_d;
6        cpuTaskList[index] = global_blockId;
7        atomicAdd(lastIndex_d,1);
8        atomicExch(cpuTaskList_lock_d, -1);     //release lock here
9    }
10   __syncthreads();
```

Listing 3.12: *Atomically appending into the task queue.*

For each block, only one thread (with thread ID 0) performs this operation as new task shall be pushed only once per block. On line 4, the `AtomicCAS` call halts the execution until the block has successfully written its *block ID* into the lock variable `cpuTaskList_lock_d`. If the lock is obtained, meaning that *Atomic Compare&Swap* succeeded, the block has entered the lock-protected critical section. Writing new task to the task queue consists of the following operations:

1. Write one's *block ID* in the task queue (array `cpuTaskList`) to *last index* stored in variable `lastIndex_d`. (line 6)
2. Atomically increase the value of `lastIndex_d` so that the next task is pushed to the next index. The operation is atomic so that first, it will not get reordered out of the critical section by the compiler, and second, the write operation is stored and not kept in a register instead. (line 7)
3. The `cpuTaskList_lock_d` lock is released (set to -1) so that the next block can enter this critical section now. (line 8)

Finally, the block-wise synchronization point `__syncthreads()` ensures that all threads of the block (even those in different wavefronts) halt the execution until the lock barrier is passed by thread 0.

Per-block channel

The second, alternative, implementation uses multiple channels for CPU-GPU communication – one for each block. It is shown in fig. 3.9b. The main advantage of this approach

is that it requires neither a lock and critical section constructs nor any kind of atomic operations in general. From the GPU side, posting a task for the CPU is simpler:

```
1    if(threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0)
2    {
3        int global_blockId = blockIdx.x + (blockIdx.y * gridDim.x) + (blockIdx.z *
             gridDim.x * gridDim.y);
4        cpuTaskList[global_blockId]=1;
5    }
6    __syncthreads();
```

Listing 3.13: *Posting a task for the CPU.*

In this implementation, again only thread 0 does the work and other threads of the block wait at the `__syncthreads()` barrier. It writes 1 into its dedicated index (in contrast to initial value -1), which means that a new task is available. The *block ID* is no longer necessary as the particular index is being written to only by one block. If there was a need to distinguish multiple different tasks, updating different bits of the variable would enable that. Alternatively, each block could have an array associated to it, instead of a single index.

On the receiver side (CPU), each index of the array has to be checked to see if one of the blocks posted a task. This means that the number of locations a CPU has to check is linear in the number of blocks, not constant as in the task queue. This characteristic puts more load on the CPU, and there may also be longer delays before a task is retrieved. Nevertheless, as the number of blocks per GPU will not be very large, in our case not more than 27, this overhead should remain reasonable.

Comparing the two approaches – task queue and per-block channel – the former can potentially suffer from poor performance of GPU-wide atomic operations, while the latter from the CPU spending too much time checking if a task was pushed. The question remains which of the overheads is larger. The performance is compared in section 4.5.2. Another notable difference is that, if multiple tasks are waiting to be processed, the task queue version processes them in FIFO (first in first out) order, whereas in per-block channel implementation, the task from a block with lower *block ID* is processed first. For some use-cases, this difference may also influence the performance.

Request completed notifications

Upon positively testing for receiving an MPI request, as mentioned in section 'CPU waits for events', the CPU informs the GPU regardless of the implementation. This takes place through another array – `requestArray`. Again, there is an index assigned to each GPU block. The blocks spin-wait until the array reaches certain threshold value – which is assigned to it by the CPU. The array (`requestArray` in code snippet 3.14) is declared `volatile` so that it is ensured that the variable is loaded anew each time and that the GPU block does not test an outdated piece of data from its register file. Again, the `__syncthreads()` barrier ensures that none of the threads of the block leaves the wait segment before it is supposed to.

```
1    if(threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0)
2    {
```

```
3         int global_blockId = blockIdx.x + (gridDim.x * blockIdx.y) + (gridDim.x *
              gridDim.y * blockIdx.z);
4         while(shared_var_threshold > requestArray[global_blockId]) { } ;
5     }
6     __syncthreads();
```

Listing 3.14: *GPU block waits for notification from the CPU.*

3.5 Memcpy version

The next adaptation of the taskqueue approach aims at exploiting faster accesses to GPU's local memory. This implementation has 2 send buffers – `send_buffer_arr` on pinned host memory and `send_buffer_arr_d` on device memory. The two buffers are in contrast to using only pinned host memory *send buffer* in the original taskqueue approach from the previous section. In brief, the variation presented in this section uses memory copy operations to copy *send buffers* to and from the device to make them available for the GPU's packing routine and CPU-handled MPI communication, respectively. The logic was not extended to the *receive buffer* and the unpacking function as the workload in the unpacking function is too small to benefit from this.

This implementation will be referred to as *memcpy*, while the zero-copy access version presented in the last section will be called simply taskqueue.

3.5.1 Workflow

The memcpy adaptation shares majority of the workflow with the taskqueue approach, and the memory copy logic is added on the top. Therefore, the focus here will be on describing the aspects that are different compared to taskqueue approach. The workflow and the interactions between the host and device are shown in fig. 3.10. In addition to what was presented in the workflow of taskqueue approach, asynchronous Host-to-Device (H2D) and Device-to-Host (D2H) memory copy operations and functionality managing them were added.

CPU launches kernels and issues memcpy

The CPU starts the iteration, equally to taskqueue, with the kernel launch. However, after the kernel launch, a series of asynchronous H2D memory copies is issued – for each block, its *send buffer* is copied from its location on the pinned host memory to an alternative location on the device memory. This memory copy is asynchronous to the host, so after calling the operation, the program can continue. What follows is posting `MPI_Irecv` operations that take care of receiving data in the *receive buffers*. All in all, the only change until this point has been the addition of the *memcpy* operations.

GPU first waits for memcpy

Most of the workload of the GPU is similar to the one of taskqueue. The largest difference comes at the beginning – before the GPU starts packing, it first waits until the device-allocated *send buffer* is copied over. The implementation of the memory copy operation is

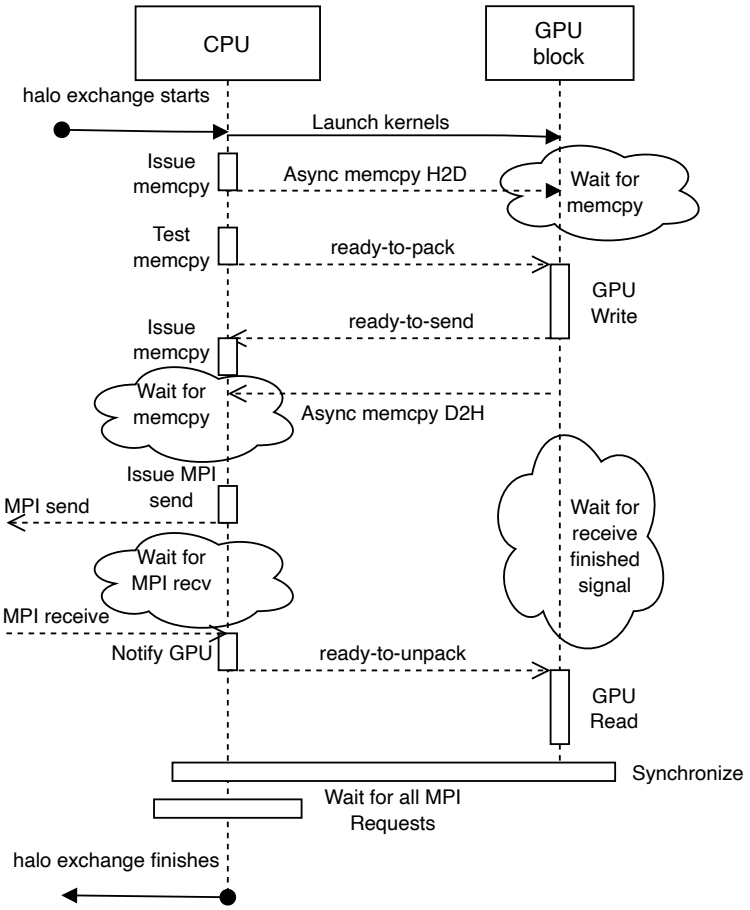


Figure 3.10: *Interactions between the CPU and a GPU block in one iteration.*

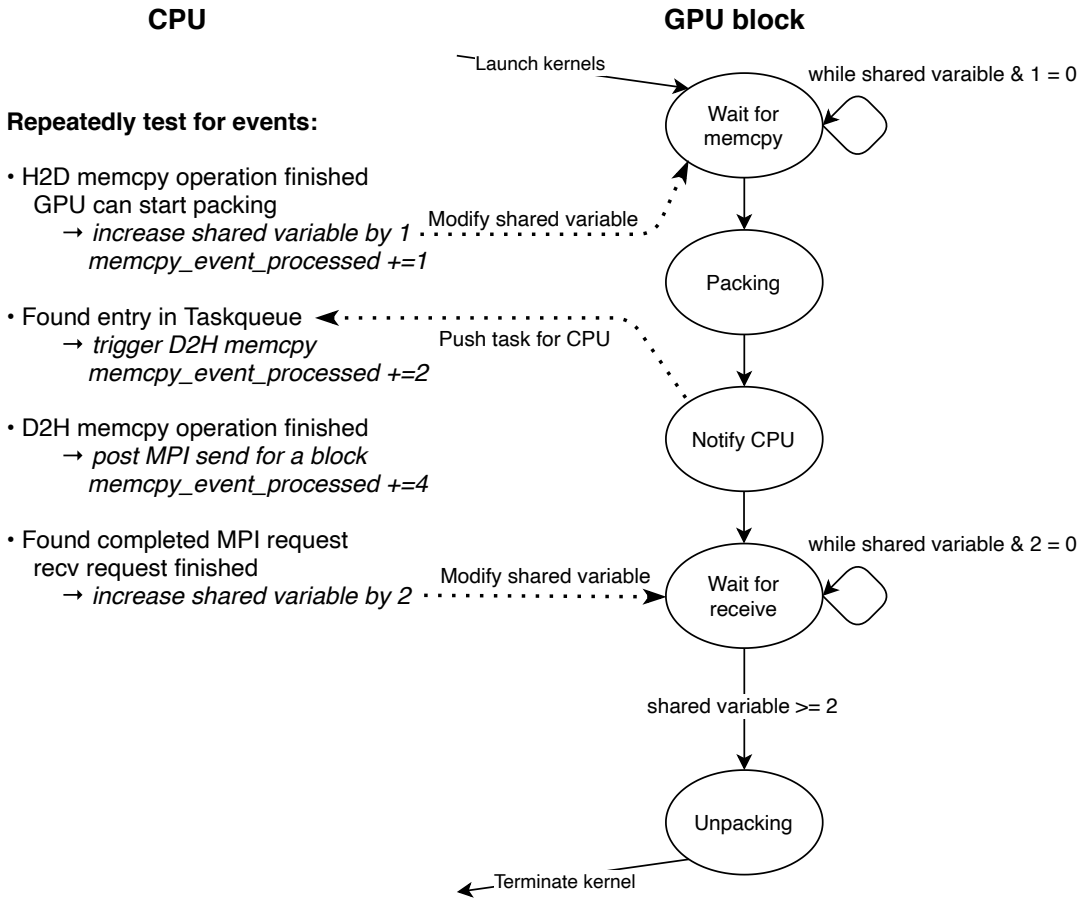


Figure 3.11: *FSM of one halo exchange iteration and one GPU block for memcpy variant.*

described in section 3.5.2. This results in a slightly updated FSM concerning the workflow of each GPU block. It is shown in figure 3.11. Compared to the taskqueue workflow (cf. figure 3.7), memcpy workflow adds one state at the beginning; before the packing begins, the GPU block waits for the memcpy operation to finish.

Once the memory copy operation has finished, the GPU block starts packing. In contrast to taskqueue, the packing function operates over a device-allocated array – `send_buffer_arr_d`. The packing function for device buffers – `GPU_Write_device_mem` – is presented in section 3.2.4. All in all, it is identical to the general packing function with the only exception of using different buffer that is written to. In this scenario, the CPU uses memcpy operation to access the data and does not access the buffers directly. Therefore, no memory fence operations after packing are necessary. The rest of the workflow is analogous to taskqueue – the GPU block pushes a task for the CPU, waits for a notification from the CPU, and unpacks the received data (on pinned host memory).

CPU waits for events

After having done the initial tasks, the CPU waits for and processes events. On the top of taskqueue's one event from the GPU (packing completed) and one event from MPI (receive operation completed) for each block, there are two additional events regarding the memory copies, namely completion of H2D and D2H memcpy operations. Therefore, there is $4 \times num\_blocks$ events in total. The workflow is shown in fig. 3.12.

There were several updates to which events are monitored and processed, and what functionality is triggered upon occurrence of these:

- When the initial H2D memcpy is finished, the CPU notifies the GPU that it can start packing.
- An event in the taskqeue does not trigger MPI communication anymore. Instead, it triggers asynchronous memcpy from the device to host.
- When a D2H memory copy operation has finished, the MPI send operation can be posted.
- As before, upon registering an incoming message, the CPU notifies the GPU that it can start unpacking.

When focussing on one particular block, first an event that denotes completion of memcpy H2D operation notifies the GPU that it can start packing. Then, an event from the GPU triggers D2H memcpy. Next, another event that denotes completion of D2H memcpy operation triggers sending the *send buffer*. Independently on these events, the completion of `MPI_Irecv` operation may create another event at any time.

Once all events are processed, the workflow for finalizing the iteration is the same as the one of the taskqueue approach. On the top of that, array `memcpy_event_processed`, which already appeared in fig. 3.11, is reset. The purpose of this array is keeping information of the state of each block's processed events, and will be discussed later.

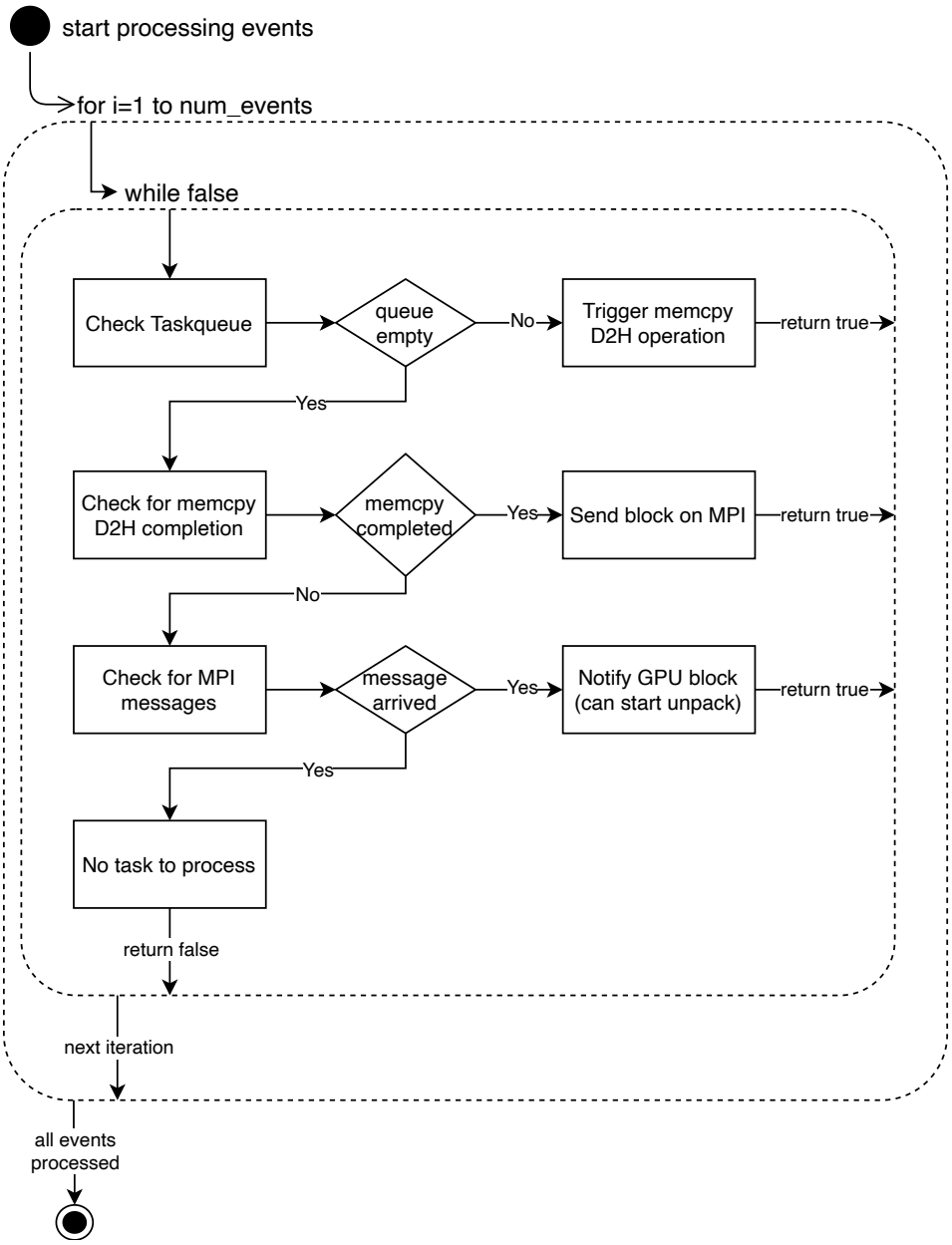


Figure 3.12: *Workflow of a CPU processing events in the Taskqueue memcpy approach.*

3.5.2 Memory copy logic

The following functionality needed to be added in order to design and implement memcpy version of taskqueue:

1. Allocate and initialize additional device buffers and all accompanying data structures.
2. Copy send buffer from host to device asynchronously.
3. On the device, wait until H2D memcpy has finished.
4. Copy send buffer from device to host asynchronously.
5. On the host, decide whether D2H memcpy has finished.

The boundaries to designing the solution were set by HIP asynchronous memory copy capabilities. HIP offers an API call very similar to the one known from CUDA – `hipMemcpyAsync`. From the perspective of the host CPU, the operation is asynchronous, and therefore has analogous attributes to non-blocking MPI calls (`MPI_Isend` or `MPI_Irecv`) for MPI communication. In particular, non-blocking check of the status of the operation must be possible. That is due to the fact that the workflow from 3.12 works identically for all decisions – *if operation is finished, do something, else move on to the next check.*

HIP streams offer means to isolate a subset of events so that all operations (such as memcpy) on a stream will be executed in the same order they were posted, but completely asynchronously to operations in other streams. HIP (as well as CUDA) programs can be designed so that a functionality that enables non-blocking querying the status of a stream – an equivalent to `MPI_Test` – is available. API call `hipStreamQuery(stream)` is used for that. It returns true (`hipSuccess`) if there is no pending or queued workload in a particular stream.

Memcpy implementation uses this property. Memory copy operations regarding each block have their own stream as a dedicated channel for memory copy operations, which is, however, asynchronous from all other blocks' streams. This way, `hipStreamQuery` can determine if there is a pending memcpy operation on a particular block or not. This functionality is available only for the host. In order to use this functionality to trigger packing, the GPU blocks have to be able to find out about a completed H2D memcpy operation on their *send buffers*, hence the host has to notify the device when it captures such an event. When a memcpy operation is completed, the host uses the same channel to notify the GPU as it uses when a message is received – an index of `requestArray` array. Value 2 (10 in binary) allows the GPU block to start unpacking. Value 1 is used to denote completed memcpy operation. The GPU can verify each of these events by masking off different bits of the `requestArray` index with logical AND operation – last bit informs about the memcpy operation, second to last bit about *receive buffer* reception (as it is the case for taskqueue approach as well). The GPU, uses the same mechanism to wait before beginning with packing and unpacking. It only spin-waits on a different bit of the same variable.

On the CPU side, there is a for loop over all blocks that performs `hipStreamQuery` call to identify if some of the operations completed. However, the piece of information

given by `hipStreamQuery` is not very specific – it can only tell whether there is a pending memcpy or not. Therefore, the CPU needs to maintain the information about state of each block to correctly react to result of `hipStreamQuery` call. The state is kept in array `memcpy_event_processed`. The initial value is 0 and it works the following way for each block:

1. Value 0 indicates that H2D memcpy is in progress. If `memcpy_event_processed[block] == 0`, perform a `hipStreamQuery` to check if the memcpy operation has finished. If so, increase `memcpy_event_processed[block]` by 1 and notify the *block* that it can start unpacking.

2. Value 1 indicates that packing is in progress. During this phase, no `hipStreamQuery` is necessary as no memcpy operation is posted. When a CPU receives an event from the GPU (packing finished), it increases `memcpy_event_processed[block]` by 2, and starts D2H memcpy.

3. Value 3 indicates that D2H memcpy is in progress. If `memcpy_event_processed[block] == 3`, perform a `hipStreamQuery` to check if the memcpy operation has finished. If so, increase `memcpy_event_processed[block]` by 4 and trigger MPI send operation.

4. A value larger than 3 indicates that both memory copies for the block finished, so there is no need to call `hipStreamQuery` anymore.

The `hipStreamQuery` call is rather costly, so it should should not be performed unless necessary. Variable `memcpy_event_processed[block]` identifies states when the status needs to be obtained and when not.

Concerning HIP streams, there is a separate stream created for the kernel – it should neither be blocked by any memcpy operation, nor should it block any of them.

```
1    hipLaunchKernelGGL(Write_GPU_Read_GPU, dim3(grid), dim3(block), 0,
         compute_stream, params...);
```

Listing 3.15: *Kernel launch on a separate stream.*

3.6 Other optimizations

While working on the topic and considering possible solutions, there were other approaches analyzed as well, however not implemented. This section contains a short overview of other directions that could present an alternative to the solutions presented in this chapter, and possible improvements to the already presented solutions.

3.6.1 Multiple threads on the CPU side

All implementations use multiple threads on the GPU side, but the CPU is implemented in a single thread fashion. As long as the CPU is idle for most of the time and manages to process all events immediately, it is not an issue. However, in case the CPU does not

manage this, it damages the performance of the overall solution. If that is the case, using multiple threads could solve the problem.

The existing implementations contain profiling functionality that gives information about the idleness of a CPU. It counts how often and how many times in a row an event is processed right after a previous one. In other words, how many events in a row are processed before a CPU does not find an event. That gives an overview of the load on the CPU. It does not provide any information about how many tasks are waiting to be processed. Obtaining that information would have performance impacts. Instead, one can see how often the CPU is idle after having processed an event. If this metric is high, using multiple threads would be an option that may improve the performance.

There are multiple possible realizations, however the first measure would likely be splitting scanning the events to different threads. That means that one thread would check the taskqueue for new events from the GPU and issue MPI send operations, while the other thread would repeatedly test for completion of MPI receive operations and inform the GPU. (In memcpy implementation, an additional thread would take care of memcpy operations.) Should that still not be performant enough, splitting each domain would be the next step. For instance, there would be multiple threads checking and processing the events from the task queue. In such a scenario, the operations would have to be made atomic on the CPU side as well as multiple CPUs would operate at the front of the queue. In per-block channel implementation, the possible division is that each thread would manage a certain subset of GPU blocks, hence array indexes. This solution could be implemented without using locks.

When running a program where multiple threads of one process take part in MPI communication, a suitable level of thread support needs to be specified (see chapter 12.4.3 in [12]).

3.6.2 Pipelining packing, memcpy, and send operations

The taskqeue approach relies on some blocks finishing packing sooner than others. If the packing operation finishes for all blocks at the same time, no overlapping of packing and MPI communication will be achieved, therefore the expected performance gain would shrink drastically. Pipelining could preserve the performance gain in such scenarios as well.

The concept is presented in figure 3.13 (cf. figure 2.5 with sequential workflow). There, the packing and unpacking workloads are split into chunks that get processed sequentially, but while one chunk is packed, the previous one can already be put on the network. This approach ensures overlapping of computation and communication even for a single block, thus in every case. On the other hand, it might suffer from dealing with larger number of operations, which means that all overheads would be counted in multiple times – for each chunk. Moreover, in order to implement this solution, packing and unpacking function would have to be adapted as well, so that it creates a task every time a chunk is ready and the task contains information on which chunk is ready to be sent.

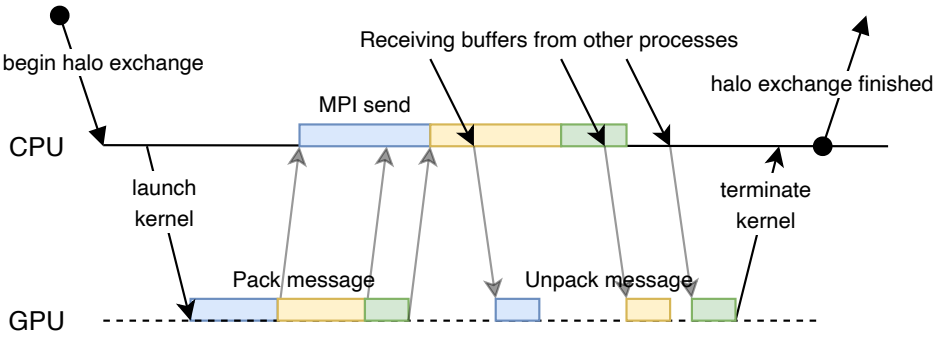


Figure 3.13: *Splitting buffer to multiple chunks to enable pipelining of packing, MPI communication, and unpacking in case the packing functions take the same time to finish for all blocks.*

3.6.3 Minor optimizations

There are also some alternatives to the way the data is stored. When using arrays for implementing the taskqueue approach, the succeeding indexes of these arrays are located on the same cache line (unless the index is at the cache line boundary). As the indexes are implemented as int of size 32 bit, 16 of them fits in one 512-bit cache line. When multiple GPU blocks write to the same array, they might experience false sharing. In such cases, having one element per cache line would prevent it from happening.

3.7 Platform-relevant implementation details

Apart from timestamping, which is different for each platform, there were other differences between the implementations for AMD and NVidia GPUs. This statement could be rephrased such that some functionality did not work on the AMD side as it should have, therefore an alternative solution had to be found.

3.7.1 Linking issue in compilation on AMD

When compiling for our AMD testbed, the *hipcc* compiler (*hip-clang* in the background) was not able to link the device functions across multiple files. While on NVidia side, this was working as expected when using `-dc` compiler flag, it was not the case for AMD compilation with the `-fgpu-rdc` flag, which should be similar in functionality according to available documentation.

The problem is not present when a CPU launches a kernel (`__global__` prefix) whose declaration and definition is in another file. However, calling a function from the GPU kernel itself (calling a `__device__` function) that is declared in another file is already problematic.

To circumvent this issue on AMD devices, the source code from all involved files was put together into one large ”spaghetti code” file before compilation. As a result, there was no linking necessary there, and the code could be compiled correctly.

3.7.2 `hipMemcpyAsync` deadlock on AMD

When implementing one of initial versions of memcpy approach (section 3.5), a deadlock was observed. In the then solution, asynchronous memory copy and kernel launch were called in this order. Both operations ran on separate streams, therefore they should have been asynchronous with respect to each other. Second, `hipMemcpyAsync` should always be asynchronous to the host, even if operating on the same stream as a running kernel function. While this approach worked as expected on the NVidia testbed, it resulted in in a deadlock on our AMD testbed.

The problem likely lied in a kernel blocking the `hipMemcpyAsync` operations on different stream. It was observed that the first few blocks received their buffers while others were blocked. This likely happened because the beginning memcpy operations were performed before the kernel was scheduled and launched and therefore before it was blocked. The kernel function was spin-waiting on the piece of data that was just being copied over. That could have played a role in causing the deadlock. However, according to HIP documentation [2], it must not be the case.

The workaround on the AMD side consisted of swapping the order of kernel launch and `hipMemcpyAsync` operations. With this simple modification of the code, the deadlock disappeared. Nevertheless, the initial logic was changed for other reasons, and this issue is not present in the final solution anymore.

3.7.3 Crashing `atomicInc` operation on AMD

Although this issue does not affect the final solution because the functionality using `atomicInc` was abandoned, it is worth mentioning. Using `atomicInc` atomic operation causes failure of our AMD testbed as well. When running code that contains this atomic function, not only the program crashes, but the whole node needs to be rebooted in order to use the GPU again. Yet the atomic function `atomicInc` is included in HIP documentation as supported.

The issues mentioned here were observed on our AMD testbed at the time of writing this thesis. It is possible that on different hardware, or using different versions of software packages and firmware, the functionality will work properly. Hopefully, future versions of HIP and ROCm platform will fix the defective functionality or delete it from the documentation.

4 Evaluation

The various implementations from previous chapter listed in tables 3.1 and 3.2 for baseline and taskqueue approaches, respectively, were tested and their performance was evaluated. This section will cover the overall performance results as well as another insights in the workflow of the algorithms.

4.1 Testbeds

There were two testbeds from LRZ (Leibniz-Rechenzentrum)[1] to run performance measurements – one featuring AMD GPUs and other with NVidia GPUs.

The former is the AMD part of the BEAST cluster. It uses an AMD EPYC 7742 CPU built on AMD's Zen 2 microarchitecture. The CPU has 64 cores on 2 sockets. There are two AMD Radeon MI-50 GPUs running on Vega20 (GCN 5) GPU architecture with a total of 3840 streamprocessors. The GPU is split into 4 Shader Engines and 60 Compute units. It has a 32 GB HBM2 memory and up to 6.6 Tflops (double precision) performance. The GPUs are connected via a PCIe 4.0 x16 bus to the CPU. OpenMPI 4.0.4, and HIP 3.5.1 were installed on the system. The *clang* version for the *hip-clang* compiler is 11.0.0.

The NVidia testbed is LRZ's DGX-1 machine [2]. There are two Intel Xeon CPU E5-2698 CPUs built on Intel's Broadwell microarchitecture, each with 20 cores (40 hardware threads). It contains 8 NVidia V100 GPUs with NVidia Volta architecture with a total of 5120 CUDA cores. Each of the GPUs has 32 GB HBM2 memory and 7.8 Tflops (double precision) performance. OpenMPI 2.1.1, and HIP 3.8.0 were installed on the system. The *nvcc* version used by the *hip-nvcc* compiler is 10.1.

The dimensions of the tests were adapted to fit both testbeds, hence to fit BEAST as a smaller system. It was possible to use both GPUs and launch the MPI processes on different sockets of the CPU. The same setup – one-process-per-socket and 2 GPUs – was used also for DGX-1.

4.2 Test setup

The overall performance tests were carried out on both testbeds with identical setups. Table 4.1 presents sizes of each buffer – the column *Array size* denotes the number of elements in *send* and *receive buffers*. As the buffers consist of 8-byte data type *double*, the column *Buffer size* shows the size of the buffers in bytes – it is the message size that is to be transported via MPI.

[1] https://www.lrz.de
[2] https://doku.lrz.de/display/PUBLIC/DGX-1

Block ID	Array size	Buffer size	Block ID	Array size	Buffer size	Block ID	Array size	Buffer size
0	1	8 B	9	5000	40 kB	18	10 000	80 kB
1	15 000	120 kB	10	20 000	160 kB	19	25 000	200 kB
2	30 000	240 kB	11	35 000	280 kB	20	40 000	320 kB
3	45 000	360 kB	12	50 000	400 kB	21	55 000	440 kB
4	60 000	480 kB	13	65 000	520 kB	22	70 000	560 kB
5	75 000	600 kB	14	80 000	640 kB	23	85 000	680 kB
6	90 000	720 kB	15	95 000	760 kB	24	100 000	800 kB
7	105 000	840 kB	16	110 000	880 kB	25	115 000	920 kB
8	120 000	960 kB	17	125 000	1000 kB	26	130 000	1040 kB

Table 4.1: *Buffer sizes of different blocks used in the measurements.*

Each presented measurement consists of 13 iterations – the first 3 are used as "warm-up" and not counted for the measurement (more details in section 'Stability of the measurements' later in this chapter), the remaining 10 iteration constitute the measured segment. Each presented measurement result consists of 40 such tests. The data is collected from *rank 0*. The total and iteration times are almost identical for both MPI processes, as each iteration ends with `MPI_Barrier`. Concerning other used metrics, such as packing times per block, these values may vary, however no significant differences between devices were observed. Also, these metrics only serve to understand and to reason about the overall performance of particular implementations, i.e. no general conclusions about the performance are drawn based on this data. Workflow plots are created from randomly selected test runs. As the algorithms are often nondeterministic to some extent, averaging timestamps of certain events from multiple runs in one plot could not give a precise insight in the workflow of a particular run.

If a measurement is conducted using less than 27 blocks, such as 4 or 9, the values for blocks 0–3 or 0–8, respectively, are used. Moreover, unless specified otherwise, the default domain splitting is done into 9 ($3 \times 3 \times 1$) blocks per each GPU and 128 ($128 \times 1 \times 1$) threads per block.

4.3 Timestamping

Timestamping is an inportant functionality that enables understanding the performance more in detail. The goal of timestamping is being able to precisely detect the time when a certain place in code is reached. It should be a vey fast and undemanding operation, so that its execution delays and interferes with the running system as little as possible.

Timestamps collect values regarding multiple events that occur during the runtime – they measure the duration of the whole iteration, and the duration of each segment of the code. That includes both the CPU and the GPU code segments.

The timestamping functionality is implemented in class *Timestamp*. This data structure enables storing all collected timestamps in a way that they can later be clearly identified.

Collected timestamps can be categorized in the following way:

1. **General CPU timestamps.** These are related to the workflow and measure each logical segment of the halo exchange iteration.
2. **Block-specific event timestamps.** These timestamps measure the events the CPU processes. As each of the events is related to a certain GPU block, this information about the block is also stored.
3. **GPU timestamps.** Also the GPUs take timestamps so that one can see and measure how long each segment of GPU's workload, such as packing and unpacking, takes.

On top of retrieving and storing the timestamp data, the *Timestamp* class offers functionality concerning processing the timestamps. Apart from retrieving timestamps, it contains functionality that helps to preprocess the data. That includes adjustments of CPU and GPU timestamps (see section 4.3.2), which are not comparable as raw data, and various options on presentation of the measured values, such as printing them for plotting or printing aggregate values for each iteration and overall.

4.3.1 GPU timestamps

C++ standard libraries offer a comparatively straightforward functionality for capturing current value of the precise clock register. For GPUs and HIP, the functionality is not that straightforward. In order to get the best possible precision, inline assembly code is used. There are instructions that enable retrieving the clock register value. As inline assembly is not a part of HIP and works directly on GCN and PTX ISAs for AMD and NVidia devices, respectively, separate code must be written for each of the platforms. Listing 4.1 presents function for retrieving the timestamps. There is a branch for each platform, which is selected based on the preprocessor directive – the other version must be hidden to the respective compiler, otherwise the code would not compile.

```
1  __device__ __forceinline__ unsigned long long __globaltimer() {
2      unsigned long long globaltimer;
3      #ifdef NVIDIA
4          asm volatile ("mov.u64 %0, %globaltimer;"   : "=l"(globaltimer));
5      #endif
6      #ifdef AMD
7          __asm__ volatile ("s_memrealtime %0\n s_waitcnt lgkmcnt(0)" : "=s" (
               globaltimer) );
8      #endif
9      return globaltimer;
10 }
```

Listing 4.1: *Implementation of device timestamp function.*

Each of the inline assembly lines retrieves a value from the clock register and stores it in `globaltimer`. Both platform use different instructions and different registers to retrieve the value.

4.3.2 Adjustment of CPU and GPU timestamps

The CPU and the GPU have their own clock oscillators and registers, which means that the clock registers on the CPU and GPU may contain different values at the same time. In case of NVidia devices, only a small offset was presented, while the frequency was very similar. Contrary to that, AMD GPUs' clocks run at different frequency – approximately 1/40-th of the one of the CPU. In order to compare events on the CPU and the GPU by their timestamps, it is necessary to adjust the GPU timestamps to reflect the CPU values. This is done by implementing a conversion function that would convert a GPU timestamp to an approximate of a CPU timestamp taken at the same moment. The clock oscillators maintain their frequency and are relatively stable, therefore the clock register value would increase linearly in time on both the CPU and the GPU. Having both a CPU and a GPU timestamps – *cpu_ts* and *gpu_ts*, respectively – taken at the same point in time gives enough information on how to convert the GPU timestamp at this point in time. Assuming the frequency is the same ($cpu\_ts = gpu\_ts + offset$), the conversion could be computed for any GPU timestamp. Unfortunately, there is no guarantee that this is the case. Both an offset and frequency difference has to be taken into account. The conversion would therefore follow a general linear function $y = a \times x + b$, in this case

$$cpu\_ts = angle \times gpu\_ts + offset, \tag{4.1}$$

where *angle* is the frequency ratio between the CPU and GPU frequencies – $\frac{cpufrequency}{gpufrequency}$. Knowing the $offset$ and *angle*, one could transfer any GPU timestamp to an equivalent CPU timestamp. Both values can be computed having two such points, knowing it is a linear function. The representation is shown in figure 4.1. The further apart the first and second measurements are, the more precise linear function is obtained. Therefore, the first measurement (*cpu_ts_0* and *gpu_ts_0*) is performed at the beginning of the program, while the second one at the end. Using the data points from figure 4.1, the offset and frequency ratio (angle) are computed the following way:

$$\begin{aligned} angle &= \frac{cpu\_ts\_1 - cpu\_ts\_0}{gpu_ts\_1 - gpu\_ts\_0}, \\ offset &= cpu\_ts\_0 - (angle \times gpu\_ts\_0) \end{aligned} \tag{4.2}$$

Taking timestamps from a CPU and a GPU at the same time is the trickiest part that may degrade the precision of the conversion. It is not possible to perform instructions on the CPU and the GPU simultaneously, therefore a roundtrip implementation known for example from NTP time synchronization (denoted roundtrip delay) procedure [21] is used. The workflow is presented in figure 4.2. The CPU launches a (single thread) kernel. As soon as the kernel is started, it modifies a shared (pinned host volatile) variable and writes its timestamp (`ts_start`) there. The CPU waits for the shared variable to change. As soon as it observes the change, it stores the value, resets the shared variable, and captures its timestamp – `ts_cpu`. The GPU waits for the shared variable to be reset and once it notices it, it writes another timestamp there – `ts_end`. The corresponding implementation is shown on listing 4.2. The two GPU timestamps are averaged to $\frac{ts\_start + ts\_end}{2}$, which is taken as the GPU timestamp. The precision is

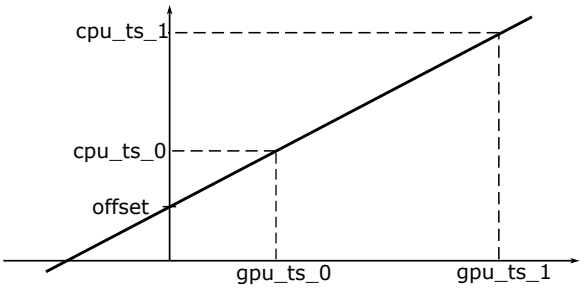


Figure 4.1: *Finding conversion function from GPU timestamps to CPU timestamps. If there are 2 points in time where both the CPU and GPU timestamp values are known, the linear conversion function for an arbitrary GPU timestamp can be calculated.*

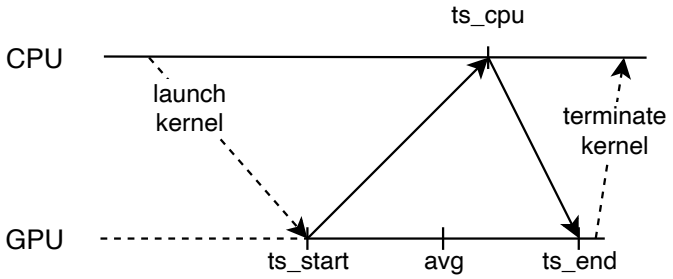


Figure 4.2: *Communication scheme in the ping-pong CPU-GPU roundtrip implementation.*

$\frac{ts\_end - ts\_start}{2}$. As the kernel launch procedure is outside of the measured region, it is not included in the interval [ts_start, ts_end]. That makes the interval shorter and thus the precision of the measurement is better.

```
1         *gpu_pingpong_ts=0;
2         hipLaunchKernelGGL(GPU_ts_pingpong, 1, 1, 0, 0, gpu_pingpong_ts);
3
4         //time-critical section starts
5         while(*gpu_pingpong_ts == 0) {};//wait for GPU to start and post its
              ts_start
6         uint64_t ts_gpu_start = (*gpu_pingpong_ts);
7         *gpu_pingpong_ts=0;
8         ts_sync_cpu[is_end*sync_measurements+i] = high_resolution_clock::now().
              time_since_epoch().count();
9         //time-critical section ends for CPU
10
11        while(*gpu_pingpong_ts == 0) {};//wait for GPU to post ts_end
12        uint64_t ts_gpu_end = (*gpu_pingpong_ts);
13        ts_sync_gpu[is_end*sync_measurements+i] = ts_gpu_start/2 + ts_gpu_end/2;
14        CHECK(hipDeviceSynchronize());
```

Listing 4.2: *Code snippet computing the conversion function between GPU and CPU timestamps.*

The whole procedure is repeated multiple times and as a result, the average values of both the offset and angle are used. Taking the computed offset and angle values, the equation 4.1 is employed to convert the GPU- to CPU-timestamp and thus get a CPU-equivalent timestamp.

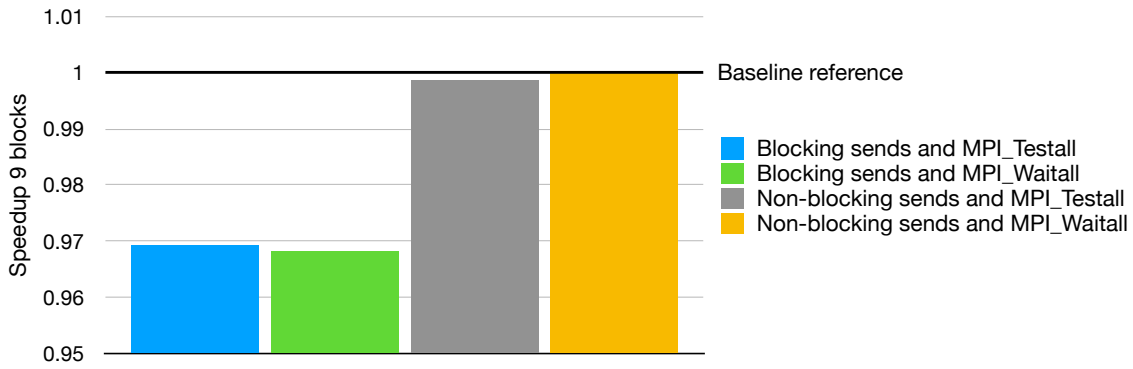


Figure 4.3: *Baseline implementations speedup on NVidia. Non-blocking communication outperforms the blocking one.*

4.4 Profiling taskqueue events

There is a very simple mechanism in place that enables gathering data concerning load on the CPU when processing events. This metric does not tell how many events there are in the system. Finding that out would slow down the execution, affect the overall performance, and skew the measurement itself. Instead, it collects data concerning positive or negative tests for event, i.e. an event was or was not found.

Two metrics are collected. The first one counts the percentage of events found after a previous test for event was also positive. If the previous test was negative, it is assured that the CPU was idle before, and the current event is therefore processed immediately. If the previous test was positive, i.e. an event was found, the event may or may not have been waiting for being processed. There is no mechanism that would find out if the event has been posted 1 ns ago, or whether it was queuing for some time. The goal is to keep the CPU idle so that processing two events in a row would occur rather sparsely. Based on the percentage of total events, this gives a hint about how busy the CPU is when processing events.

The second metric counts maximum number of events processed in a row (without a negative test in between). The information contained in this metric is similar. Still, having a high percentage (first metric) and lower number of events processed in a row indicates that the CPU is similarly busy throughout the whole iteration or measurement. Contrary to that, if there is a rather low percentage and high number of events, it indicates that there are phases where lots of events get generated, which puts high load on the CPU, while in other phases, it is rather idle.

4.5 NVidia testbed performance

4.5.1 Baseline code

Figure 4.3 shows relative performance of different baseline implementations for the default setup with 2 MPI processes and GPUs, 9 blocks per GPU, and 128 threads per block. The implementation using non-blocking MPI communication and `MPI_Waitall` calls is the

Baseline performance results on NVidia

	MPI_Waitall	MPI_Testany
Using HIP coherent memory:		
MPI non-blocking sends	3340.2 µs	3344.5 µs
MPI blocking sends	3449.7 µs	3445.3 µs

Table 4.2: *Baseline performance results on NVidia testbed. Results for each implementation with average times per iteration.*

fastest one, and will therefore be used as a reference – its performance is set to 1. Table 4.2 presents the absolute measured values. For each of the baseline implementations, it shows the average time per iteration. Analysing the results, there were several observations made:

1. The standard deviation over the set of measurements was between 2.2 % and 3.8 % of the iteration time. Thus, the measured times over the repeated runs were quite stable.

2. Regardless of using blocking or non-blocking MPI communication, the variants using `MPI_Testany` and `MPI_Waitall` show very similar performance. Both `MPI_Testany` implementations have slightly lower standard deviation compared to their counterparst (78 µs/iteration vs. 119 µs/iteration and 114 µs/iteration vs. 128 µs/iteration for MPI blocking and non-blocking communication, respectively). These two observations combined indicate that using `MPI_Testany`, the performance penalty is negligible, if any, and the implementation is similarly stable.

3. Both implementations using non-blocking MPI send operations perform better than their blocking counterparts. The difference in performance is ca 3 %, which is about one standard deviation. The better performance of non-blocking implementations indicates that there is no problem with potentially having multiple messages in progress posted with `MPI_Isend` – the network components do not get flooded with messages. On the contrary, it performs better. The potential reasons for that were discussed in section Blocking sends.

A single iteration from the reference baseline code (non-blocking with `MPI_Testany`) in figure 4.4 imitates the expected workflow (cf. figure 3.3). This plot focusses on the workflow of one iteration on one MPI process and one GPU. Each iteration starts with packing phase, where each of the GPU blocks needs different time as each *send buffer* has a different size. Packing times are indicated by the bars by the respective GPU blocks. The communication phase, which is shown in violet at the CPU level, starts after the last block finished packing. It takes the majority of the iteration time. `MPI_Testany` enables capturing the time when each of the messages was received. This is represented by the dots of respective blocks' colours above the violet line. Finally, when all blocks are received, unpacking takes place. It is again indicated for each GPU block. As mentioned in section 3.2, the unpacking workload is constant and therefore, it takes approximately the same time regardless of the buffer size. Right after that, the iteration finishes.

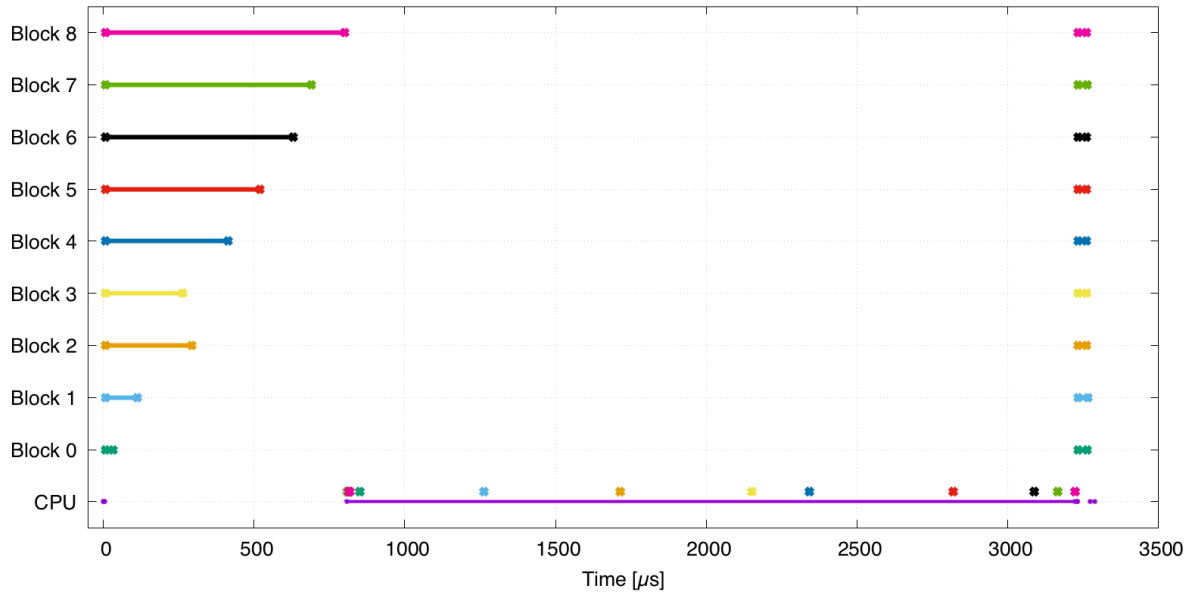


Figure 4.4: *One iteration of non-blocking baseline with* `MPI_Testany` *on NVidia. The 'Block' lines denote packing and unpacking times. The violet 'CPU' line denotes the communication phase, the dots of respective colours denote receiving a message of a particular block.*

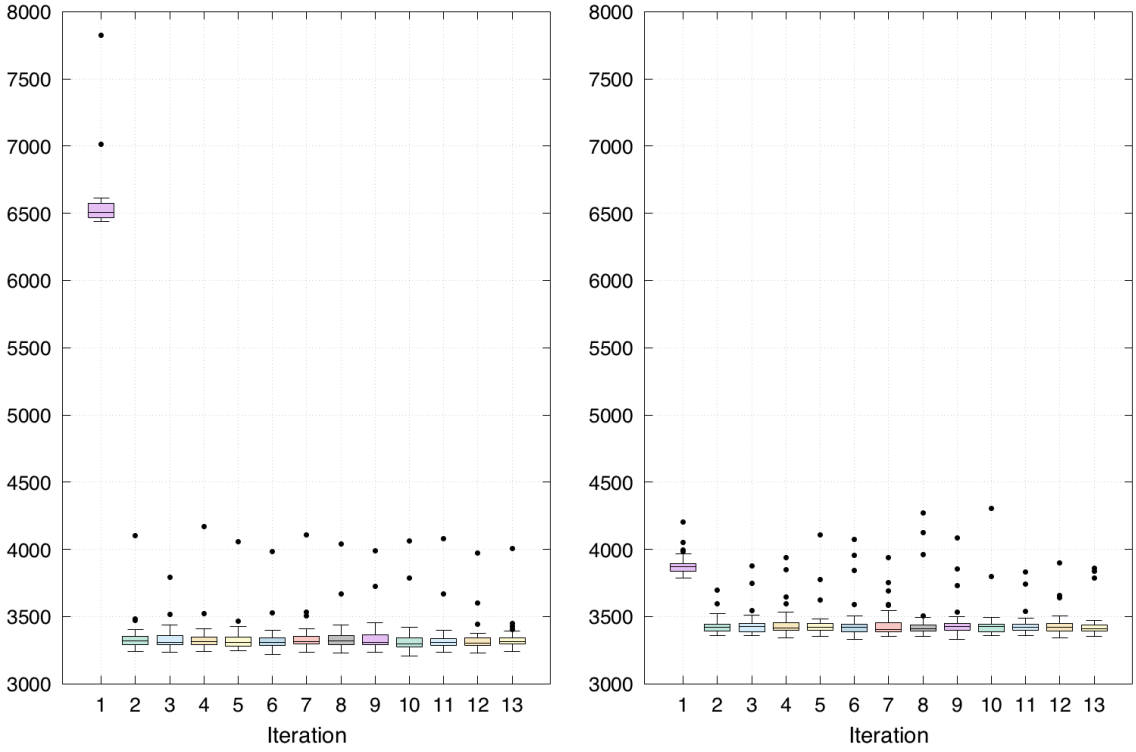
	MPI_Waitall	MPI_Testany
Using HIP coherent memory:		
MPI non-blocking sends	1 (2765.2 µs)	0.997 (2772.9 µs)
MPI blocking sends	0.958 (2886.9 µs)	0.955 (2894.7 µs)

Table 4.3: *Baseline measurements identical to the ones presented above, however taken on a different day. It contains the measured speedup, and in the parentheses is the average time per iteration.*

Stability of the measurements

Table 4.3 shows results of identical performance measurements to the ones presented in table 4.2 done on the same machine, but on a different day. The absolute values presented in both tables differ significantly, but the trends observed in the previous table are still present. The absolute times were ca. $550-575\,\mu s$/iteration ($16-18\,\%$) less for all corresponding implementations, which indicates that the performance of the NVidia testbed differs in time. This is the reason why all relevant performance measurements were taken in a short time interval. On the other hand, these results show the same trends as described above – the difference between `MPI_Testany` and `MPI_Waitall` is negligible, and the blocking communication performs worse than non-blocking.

Figure 4.5a is a boxplot showing summary data of the individual measurements on the baseline reference (non-blocking with `MPI_Waitall`). The boxplot shows all 13 iterations of one run, and for each one it evaluates the measured times. It clearly shows that the



(a) *Baseline non-blocking* `MPI_Waitall`. **(b)** *Baseline blocking* `MPI_Waitall`.

Figure 4.5: *Boxplots showing stability of the iterations on NVidia testbed. Y axis presents times per iteration in* µs. *The first iteration takes longer, and is a lot less stable.*

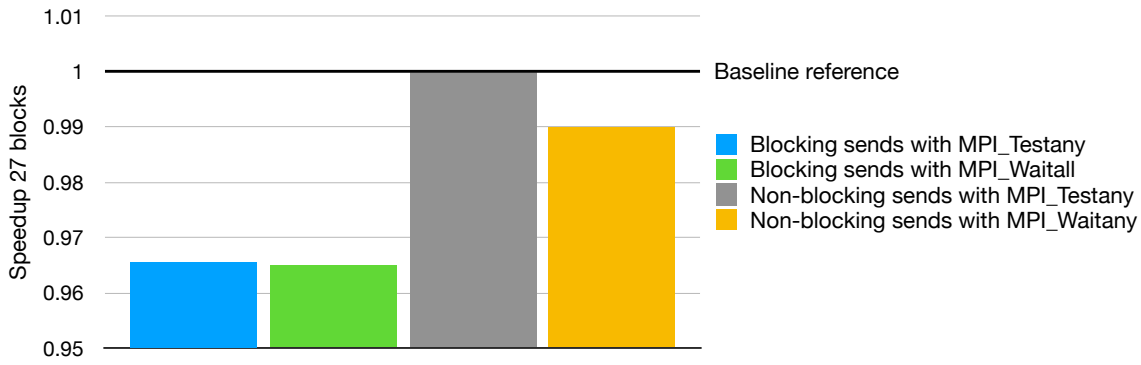


Figure 4.6: *Baseline implementations with 27 blocks – speedup on NVidia. Non-blocking communication still outperforms the blocking one.*

first iteration takes significantly longer time to finish. This is most likely caused by MPI handshaking protocols when sending messages from the source to the target MPI process for the first time. As the handshaking does not need to take place multiple times, other iterations run a lot faster, and are a lot more stable. There are still some outliers in other iterations, but as there are mostly only a few, it does not skew the average values significantly. The other non-blocking implementation (`MPI_Testany`) performs very similarly. The first iteration takes 6590 µs, while the average of iterations 4–13, which is the measured segment, is 3345 µs.

Blocking implementations suffer from a significantly lower performance penalty in the first iteration. The second boxplot – 4.5b – shows the measurement data of a blocking implementation (again using `MPI_Waitall`). It takes 3883 µs (average for iterations 4–13 is 3450 µs) and 3910 µs (average for iterations 4–13 is 3445 µs) for `MPI_Waitall` and `MPI_Testany` options, respectively. Apart from the first iteration, both versions are rather similar – the blocking one has slightly more outliers, but the difference is not significant. Comparing the blocking and non-blocking approaches, the values indicate that there is a performance penalty for all baseline versions, however non-blocking MPI communication suffers a lot more from the handshake protocol delay that occurs in the first iteration.

In other implementations, the different performance can be seen in up to three initial iterations, the first three iterations are not included in the measurement in order to

- assure better stability of the results,
- provide more accurate temporal results concerning one iteration, and
- not skew the results with data from the first iteration, as the goal is to observe a typical iteration.

Full problem size

With increased workload, the solutions still maintain very similar relative performance for the baseline variants. When the workload rises to 27 blocks, and each of them utilizes 128 SIMT threads, the non-blocking option still outperforms the blocking one by a similar

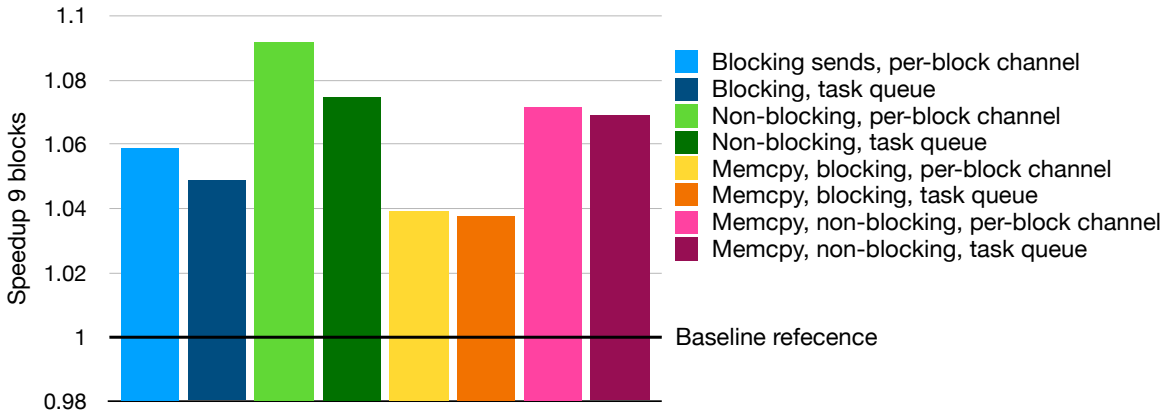


Figure 4.7: *Taskqueue implementations speedup on NVidia. Fastest baseline is set as reference, the speedups of taskqueue approaches ranges from ca. 4 to 9 %.*

margin. Figure 4.6 shows performance results for this workload. The blocking option is ca. 3 % faster on average. Contrary to the previous example, nonblocking version with `MPI_Testany` outperforms the option with `MPI_Waitall` by 1 %. This difference is most likely caused by an imprecision in the measurement, as in other measurements with the same and different workloads, these two options were performing very similarly. Moreover, the difference is less than 1/3 of `MPI_Waitall`'s standard deviation.

4.5.2 Taskqueue

Using the default workload – 9 blocks and 128 threads per block – all variants of taskqueue approach outperform the best baseline approach by 3.7 %–9.2 %. The performance of all implementations in comparison to baseline reference is visualised in figure 4.7. When computing the speedup of taskqueue blocking variants compared to the best blocking baseline, the performance gain of these implementations reaches very similar values to the ones of non-blocking implementations. The comparison is detailed on table 4.4. There, the blocking versions are compared with best blocking baseline and the non-blocking ones with the best non-blocking baseline. The performance gain ranges from 6.9 % to 9.2 %, thus is more similar for all available options. That being said, one can observe several trends concerning the performance data:

1. As well as in baseline implementations (cf. table 4.2), non-blocking communication clearly outperforms blocking communication implementations.

2. When reflecting the speedup of blocking implementations against blocking baseline, and the same is done for the non-blocking domain, the corresponding options are very similar in performance. It can be verified in table 4.4. In particular, using the option without atomic operations (per-block channel), the speedups are identical for both pinned-host memory and *memcpy* versions. Using task queue, the differences are very small as well. (0.7 % for pinned-host memory and 0.1 % for *memcpy* versions.) This observation indicates that the performance gain does not depend on the MPI

Taskqueue performance results on NVidia

	Task queue	Per-block channel
Using pinned host memory:		
MPI non-blocking sends (speedup vs. non-blocking)	3107.9 µs (1.075)	3059.1 µs (1.092)
MPI blocking sends (speedup vs. blocking)	3185.0 µs (1.082)	3154.3 µs (1.092)
Using device memory + memcpy:		
MPI non-blocking sends (speedup vs. non-blocking)	3125.4 µs (1.069)	3116.6 µs (1.072)
MPI blocking sends (speedup vs. blocking)	3107.9 µs (1.070)	3214.9 µs (1.072)

Table 4.4: *Taskqueue implementations speedup on NVidia compared to blocking and non-blocking references. Blocking options are compared to best blocking baseline, non-blocking options to the fastest non-blocking baseline. The corresponding blocking and non-blocking implementations reach very similar speedups.*

communication type; the communication workflow is likely identical for all taskqueue options and the difference in performance lies in other implementation details.

3. Pinned host memory implementation slightly outperforms the *memcpy* implementation in all cases. For this workload, it is clearly superior. On the other hand, the differences in performance gain are never more than 2 %, which indicates that both versions perform similarly well.

4. The per-block channel implementation tests yield better results than the ones of the task queue implementation. Even though the task queue implementation puts less load on the CPU (it only checks one index, see section 3.4.4 for more details), the atomic operations on the GPU side slow down the solution more than the what the gain on the CPU side is.

Overall, the results show that taskqueue approaches perform better than baseline, the non-blocking communication better than the blocking one, pinned host memory better than *memcpy* option, and per-block communication channel better single than task queue. In order to understand the reasons for that, plots of one iteration are helpful. Figure 4.8 shows the workflow one iteration.

It can be seen that the packing times for blocks 2–8 are very similar, even though the buffer sizes are different (cf. baseline plot in fig. 4.4). Compared to the baseline, there are two notable differences in the taskqueue design and workflow that could be responsible for this behaviour:

1. As there is no kernel termination at the end of the packing workload, there is an explicit memory fence operation as described in section 3.4.3 . This operation creates additional traffic in the system and causes a delay at the end of packing.

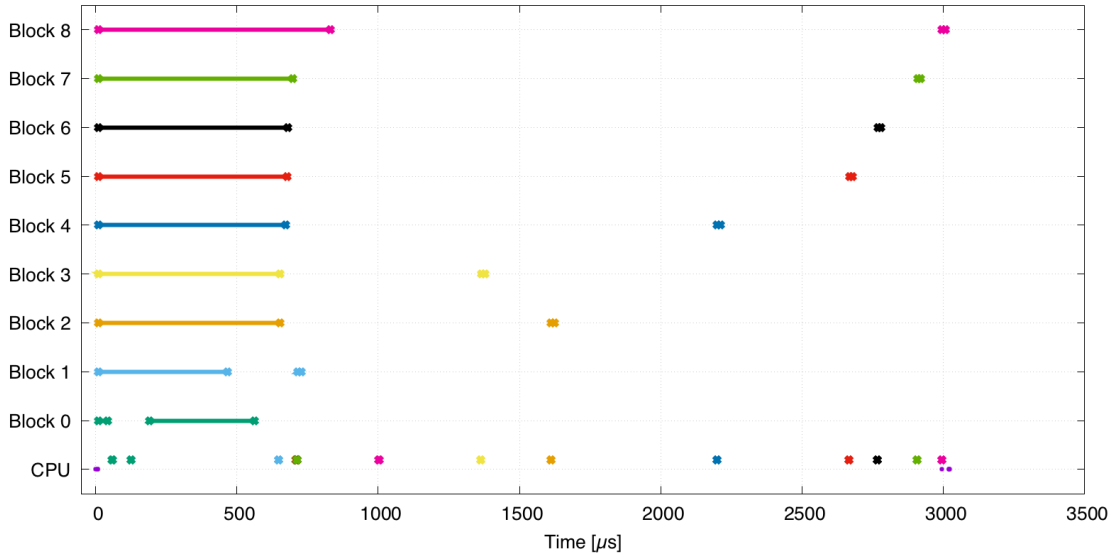


Figure 4.8: *One iteration of taskqueue non-blocking with per-block channel. The CPU processes events regarding all blocks – an event is a dot of respective block's colour.*

2. All threads of all blocks write to the *send buffer* simultaneously. These actions put load on the PCIe bus that connects the GPU to the pinned host memory, which is a part of the main memory. As this is the case for the baseline as well, and the behaviour was not observed there, this will likely not be the source of the problem. What is different, though, is the already ongoing (in both directions) MPI communication concerning *blocks 0* and *1*. MPI communication utilizes the system components as well, and may even interfere with the CUDA runtime and its mechanisms for assuring coherence of data – the source buffers were recently written to by the GPU. MPI also needs to access the main memory and its page-locked segment for its own purposes. As both processes reside on the same CPU, MPI communication will likely be implemented as memory copy operations between the *send* and *receive buffers*, which reside on the same shared memory. All of these operations put even more stress on the main memory, which is the likely cause for delays in the GPU packing.

In order to support the second claim, figure 4.9 presents the average packing times of each block in different implementations. There are three clusters of lines to be observed:

- The green one, which is almost a straight line, hides all four *memcpy* implementations. It is the fastest approach to packing, as the device buffers are written to.[3] Access to these buffers is faster than access to the CPU memory over the PCIe bus. Moreover, it is also the most stable option as the bandwidth to the device memory is presumably a lot higher than the bandwidth to the pinned host memory. Also, GPU caches are utilized, and for example cache prefetching can take place here. Apart from the

[3]Memory copies from host to device and back are not a part of the measured segment.

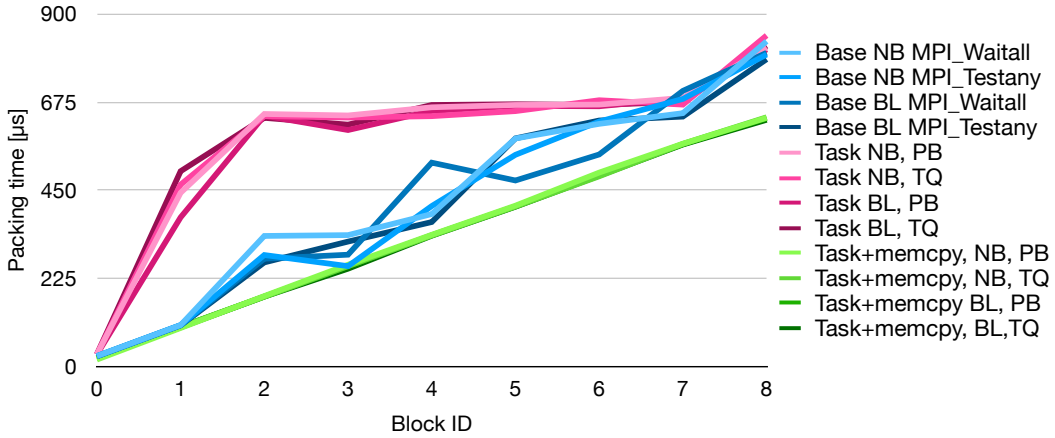


Figure 4.9: *NVidia packing times with 9 blocks. Green lines represent taskqueue memcpy options, purple lines taskqueue pinned host memory options, and blue lines baseline. (Base=baseline, Task=taskqueue, BL=blocking sends, NB=non-blocking sends, PB=per-block channel, TQ=task queue)*

packing workload, there should be no other task that stresses the device memory. Finally, as the packing functions do not start at the same time, there may be less threads packing in parallel.

- The cluster of four blue lines represents the baseline implementations. They are slower and not nearly as stable as the *memcpy* versions. Still, the packing times are comparable. The reason for the lack of stability can be that a part of the system between the GPU and the pinned host memory data (the PCIe bus or main memory itself) is getting overloaded. In such case, access time would depend on various nondeterministic aspects that decide which piece of data gets the priority of transfer.

- Finally, the slowest (pink) cluster of lines belongs to the taskqueue implementations using pinned host memory. Especially for blocks with lower IDs, it is the slowest approach by a large margin. Apart from the memory fence operation, a possible explanation of the poor performance is that there is concurrent MPI communication taking place in the system.

  Whenever a block finishes packing, there is suddenly 128 threads less that read and write data. This may be the explanation why the performance for blocks with larger buffers is not as bad compared to other implementations anymore.

The long packing times of blocks with low ID in all taskqueue implementations hinder the overall performance of the approach. The approach relies on issuing MPI communication as soon as possible. However, with the exception of *block 0*, which has a size of 8 B, the second block takes around 450 µs to finish, which means that in this time frame, no overlapping of computation and communication will take place.

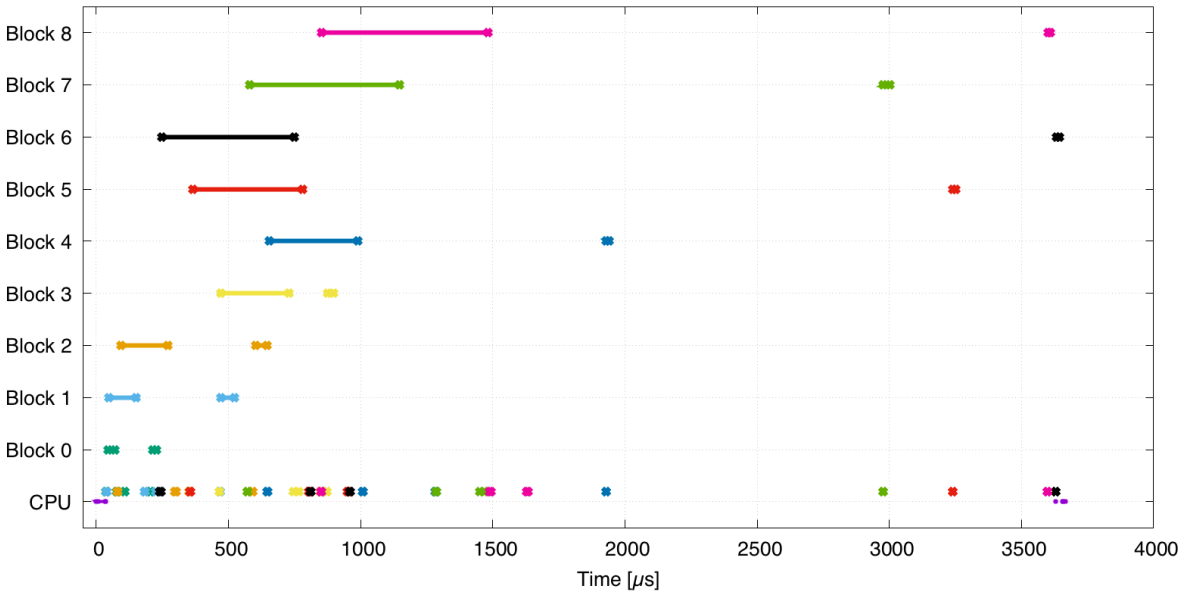


Figure 4.10: *One iteration of taskqueue memcpy non-blocking with per-block channel on NVidia. The packing starts first when the buffer is copied over, and there is twice as many events in the system.*

Memcpy variant

Figure 4.10 presents a workflow of the *memcpy* implementation (taskqueue with *memcpy*, non-blocking sends, and per-block channel). The workflow is a lot more chaotic compared to taskqueue with pinned host memory (cf. fig. 4.8). First, the packing starts at a different time for each block, which is caused by the blocks initially waiting for the asynchronous *memcpy* operation over their *send buffer* to finish; each buffer is copied over at a different time. Nevertheless, the packing times are ascending linearly, as is the buffer size, which is an expected behaviour (also visualized in fig. 4.9). As each buffer starts at a different time, not all buffers overlap in their packing time, which is a factor that further reduces the stress of the critical components of the system.

On the CPU side, there is a lot more dots there that represent the events being processed. The additional events represent completion of H2D and D2H memory copy operations. Therefore, it can be expected that the load on the CPU may be larger, especially in the beginning. The utilization of the CPU is discussed in section 4.5.4.

Figure 4.11 presents a summary of measured implementations with the default workload. The boxplot presents the measured times over each run of 10 measured iterations. Even though there are some differences in average processing time there, the stability of the results is similar.

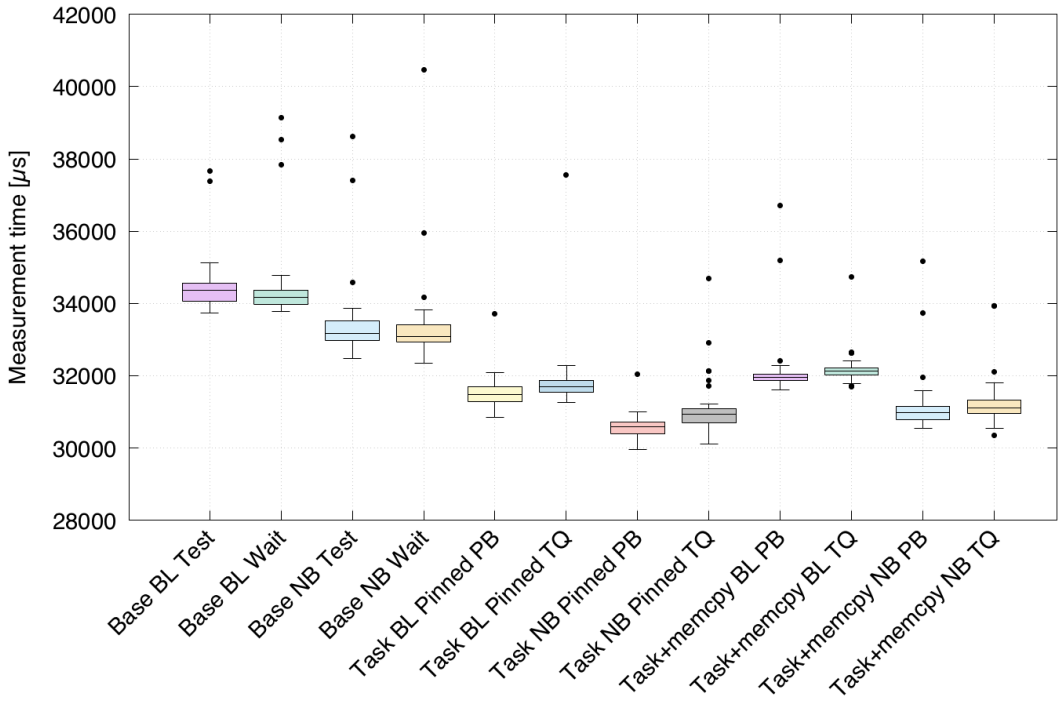


Figure 4.11: *Boxplot with summary over times of all measurements on NVidia with 9 blocks. Apart from some outliers, the stability of the results is good. (Base=baseline, Task=taskqueue, BL=blocking sends, NB=non-blocking sends, Test=*`MPI_Testany`*, Wait=Test=*`MPI_Waitall`*, PB=per-block channel, TQ=task queue)*

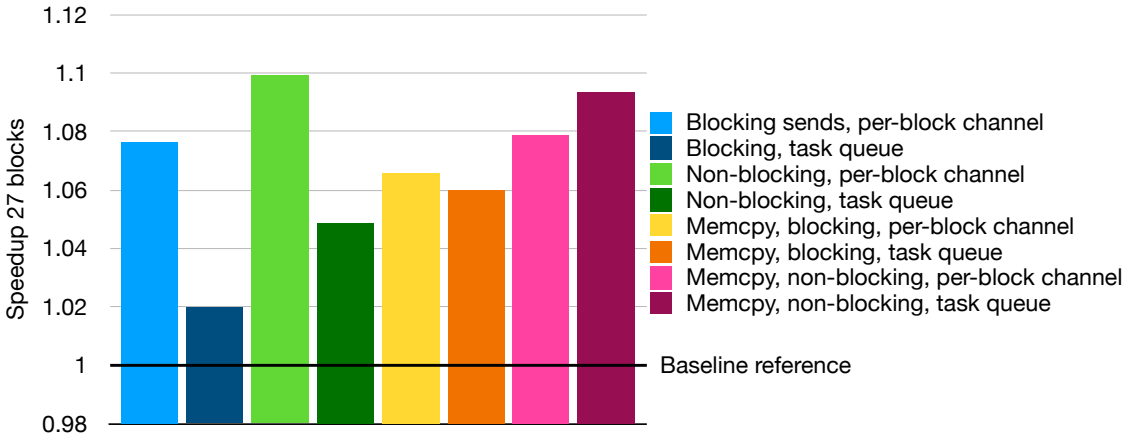


Figure 4.12: *Taskqueue implementations speedup on NVidia with 27 blocks. Fastest baseline is set as reference, the speedups of taskqueue approaches ranges from ca. 2 to 10 %.*

Taskqueue performance results on NVidia with 27 bocks

	Task queue	Per-block channel
Using pinned host memory:		
MPI non-blocking sends (speedup vs. non-blocking)	9848 µs (1.049)	9392 µs (1.099)
MPI blocking sends (speedup vs. blocking)	10 126 µs (1.056)	9593 µs (1.115)
Using device memory + memcpy:		
MPI non-blocking sends (speedup vs. non-blocking)	9441 µs (1.094)	9573 µs (1.079)
MPI blocking sends (speedup vs. blocking)	9742 µs (1.098)	9689 µs (1.104)

Table 4.5: *Taskqueue implementations speedup on NVidia with 27 blocks compared to blocking and non-blocking references. Blocking options are compared to best blocking baseline, non-blocking options to the fastest non-blocking baseline. The corresponding blocking and non-blocking implementations do not reach such similar speedups as in previous table 4.4, but the values are still similar, especially for task queue.*

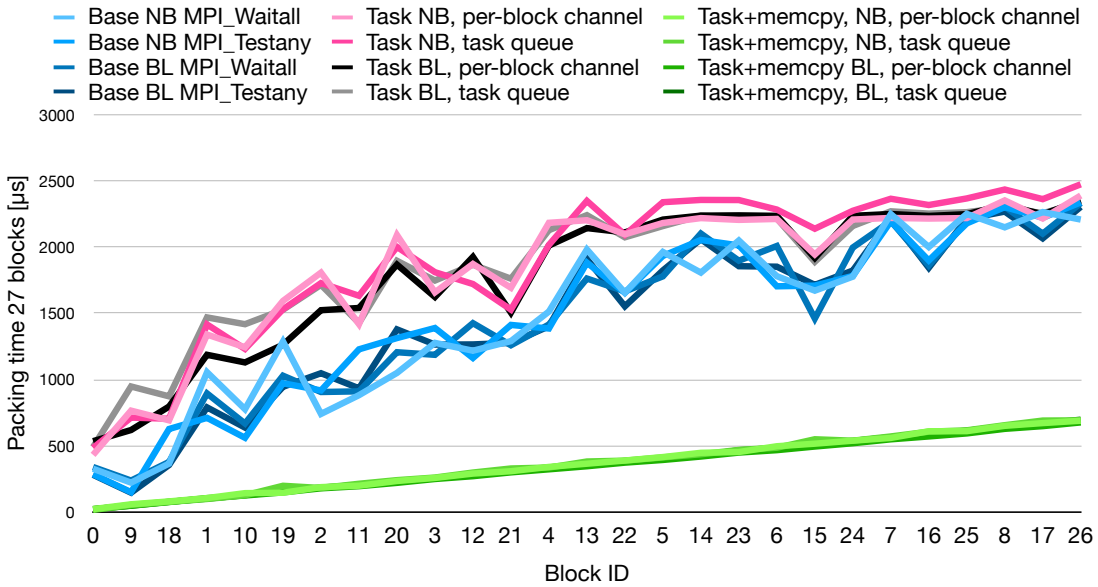


Figure 4.13: *NVidia packing times with 27 blocks. Green lines represent taskqueue memcpy options, purple and grey lines taskqueue pinned host memory options, and blue lines baseline. (Base=baseline, Task=taskqueue, BL=blocking sends, NB=non-blocking sends)*

4.5.3 Full problem size

Figure 4.12 shows the performance comparison for the approaches using the full problem size – 27 blocks with 128 threads each. The exact iteration times as well as a comparison with a blocking or non-blocking baseline are to be found in table 4.5. The trends and values in performance gain of the solutions are quite similar to the ones presented previously. The speedup is between 4.9 % and 9.9 % for non-blocking implementations and 5.6 % to 11.5 % for blocking communication options[4]. Also, the standard deviation among the measurement set grew from approximately 1 % of the average (9 blocks) to 2 %. This indicates that the variance of the measured times is larger, but still the results stay stable. The reason for that is yet even more stress on the parts of the system that started to be problematic even with the previous workload. Figure 4.13 shows packing times for the different approaches (cf. fig. 4.9).

As well as in figure 4.9, there are three clusters of lines in the plot – the green one (fastest) belongs to *memcpy* implementations, the blue one to baseline, and the pink and grey (slowest) one represents taskqueue approach using pinned host memory. Having increased the workload, the difference between the pinned host and device memory implementations grew significantly larger. While the smallest block (8 B) is packed almost immediately (20 µs) using the device memory option, taskqueue implementations with pinned host memory take ca 500 µs to pack the smallest block. As the baseline approach shows similar characteristics, and is only slightly faster, it is safe to assume that the worse performance is caused by congesting the system with pinned-host memory accesses. It did not manifest itself fully when utilizing only 9 blocks in the previous case, but in this case, the implications are more obvious.

While even the largest device-allocated buffer gets packed within 700 µs on average, implementations using pinned host memory need between 2.20 ms and 2.48 ms, which is more than three times as much. Yet overall, the taskqueue options with pinned host and device memory buffers perform rather similarly. The device-allocated buffers are faster to pack, however they also need to be copied to and from the device. Combining the two factors, the performance gain and the overhead even out so that the overall times end up being almost equal.

A rather significant difference was measured in the performance of MPI non-blocking pinned host versions – one using task queue and the other one per-block channel. The raw data behind figure 4.13 show that these two variants differ a lot in packing times for the smallest blocks. In the figure, they are represented by black line for task queue and grey line for per-block channel. While *block 0* is packed approximately after 500 µs by both, *block 9* (second to smallest, 40 kB) takes 620 µs for per-block channel and 950 µs for task queue. This indicates that task queue version can start sending the first buffer with data (ignoring *block 0* with 8 B) approximately 250 µs later[5]. This difference constitutes already over one half of the total time difference between those two.

A plot with packing times for selected implementations with a small workload using 4 blocks and 16 threads per block is presented in figure 4.14. It shows that the packing times

[4] Compared to fastest blocking baseline reference.

[5] 250 µs because *block 18* using task queue is packed after 870 µs, i.e. 80 µs faster than *block 9*.

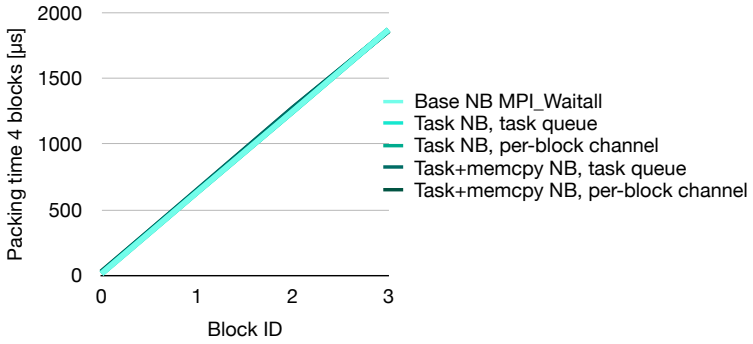


Figure 4.14: *NVidia packing times with 4 blocks and 16 threads per block. All tested options yielded very similar behavious. Comparatively longer times to previous plots are due to only 16 packing threads per block. (Base=baseline, Task=taskqueue, BL=blocking sends, NB=non-blocking sends)*

are very stable and increase almost perfectly linearly. Moreover, the packing times are very similar for pinned host baseline and taskqueue as well as for *memcpy* taskqueue. The load in this scenario was significantly lower, and the packing times were stabilized, which supports the assumption that the unstable packing times in measurements with higher workloads are indeed caused by congestion on some part of the system. The congestion causes that the data for the GPUs are loaded and stored tardily, which hinders the parallel performance.

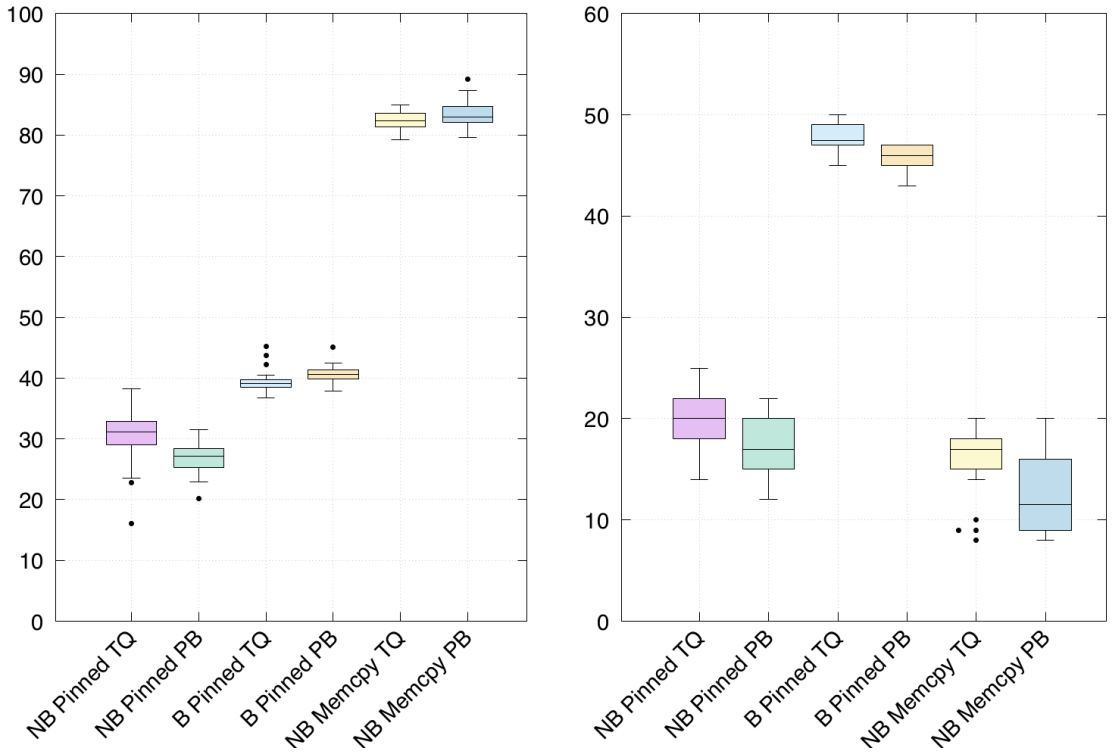
4.5.4 Profiling taskqueue variants

Profiling data helps to assess the utilization of the CPU. Based on these metrics, it can be decided whether the current CPU-sided single thread solution is sufficient, or if it became the bottleneck of the whole algorithm. Boxplots in figure 4.15 compare profiling results for chosen implementations with the 27-blocks workload. Each column refers to a different version of taskqueue.

Fig. 4.15a shows the percentage of events processed right after another event (there was no negative test for event in between). Task queue (denoted 'TQ') and per-block channel (PB) report similar values for all options. The percentage is lower for pinned host memory options (Pinned) than for *memcpy* variants (Memcpy). Non-blocking (NB) pinned host memory options are less busy than the blocking ones (B). Still, the utilization data is similar – mostly between 25 % and 45 % events were chained. Contrary to this, *memcpy* versions are obviously overloaded, processing 80 %–90 % of the events right after a previous one.

Fig. 4.15b shows the highest number of events that are processed in a row. There, the blocking implementations show much higher values than the others. This means that there may be many events waiting to be processed at once.

Based on the first boxplot, one can conclude that the *memcpy* approaches do not manage to process the events in time, and therefore the CPU hinders the performance. Splitting processing of different events to different threads could therefore be an option



(a) *Percentage of events processed right after another event.*

(b) *Highest mumber of events processed in a row (no negative test for event in between).*

Figure 4.15: *Boxplot with profiling on NVidia. The metrics are shown for different taskqueue options. Pinned host memory tends to be less busy than device memory options, and blocking send events overload the CPU in some phases of the iteration. (NB=non-blocking, B=blocking, Pinned=pinned host memory, TQ=task queue, PB=per-block channel)*

to reduce this bottleneck. The second boxplot indicates that there are numerous events created in a certain stage of the algorithm. The congestion is caused by the blocking send operations. Therefore, having a separate thread for other event types would probably not improve this metric significantly. Nevertheless, it is useful to see this metric in order to understand whether processing of the blocking sends is sufficiently fast.

4.5.5 Extreme workloads

The measurements were also performed on untypical workloads with regards to problem size definition from section 2.5.1. One deals with very large buffers – there are still 27 blocks there with 128 threads each, but the buffer sizes are 10 times larger than the default ones, i.e. the smallest buffer has a size of 80 B and the largest one has 10.4 MB. The second workload presents very small buffers of 4 blocks with 16 threads each, and the buffer sizes are 8, 120, 240, and 360 B, i.e. $\frac{1}{1000}$ of the original workload. The sizes of buffers were chosen so that they all fall under the *eager limit*, which is a buffer size limit where MPI switches between *eager* and *rendezvous* protocols. Eager messages, due to their small size, can be sent directly to the destination. Rendezvous messages are not sent until the receiver responds that the transfer may begin. As *rendezvous* messages are larger, the sender waits for the acknowledgement from the receiver. That indicates that that the receiver has a buffer prepared where the incoming message can be stored.

Very small workloads

Figure 4.16 compares the achieved speedup over the small data set. In all measured flavours, the blocking and non-blocking variants achieve very similar performance. This is likely due to the fact that all the messages' sizes are below the *eager limit*, thus are sent eagerly. It means that the blocking send operation can put the message directly on the network without communicating with the recipient. This operation can be very quick.

The taskqueue approach using pinned host memory achieves a speedup of 1.091 and 1.097 , respectively. Contrarily, the versions using device memory yielded a comparatively poor performance, performing signigicantly worse than the baseline approach. These results indicate that for such small amounts of data and a small number of concurrent GPU threads, the overhead for copying the buffers between the host and the device is much larger than the possible gain caused by using the faster and higher-bandwidth device memory. With such a small number of concurrent threads on the GPU, the host memory bandwidth is not fully utilized and therefore, the gain is minimal, if any, as figure 4.14 already showed.

The plots in figure 4.18 show workflow of one iteration for three of the measured approaches, all using non-blocking sends – fig. 4.18a presents baseline `MPI_Waitall`, fig. 4.18b taskqueue with pinned host memory and per-block channel, and 4.18c taskqueue with *memcpy* and per-block channel. As the measured times are very short, it is possible to see some of the overheads in these plots, that would normally be too small to observe. On the presented plots, packing times are very similar for each block. It is due to the fact that the workload is so small that initialization tasks, such as retrieving buffer pointers and sizes, or calculating new value to send, present the majority of the workload. Also,

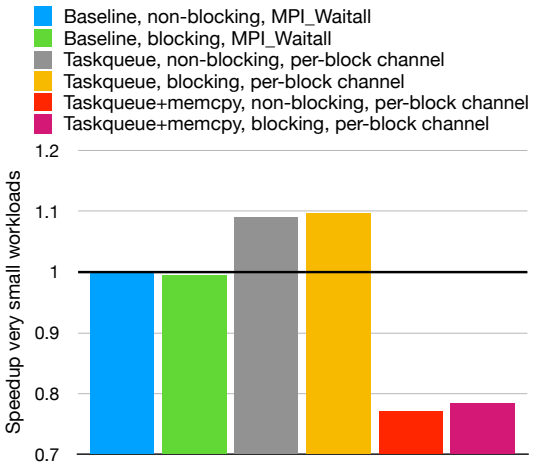


Figure 4.16: *Speedup of selected implementations on NVidia with very small workloads. Faster baseline is set as reference, the speedups of pinned host memory taskqueue approaches are around* 1.09 *, while memcpy options range between* 0.77 *and* 0.79

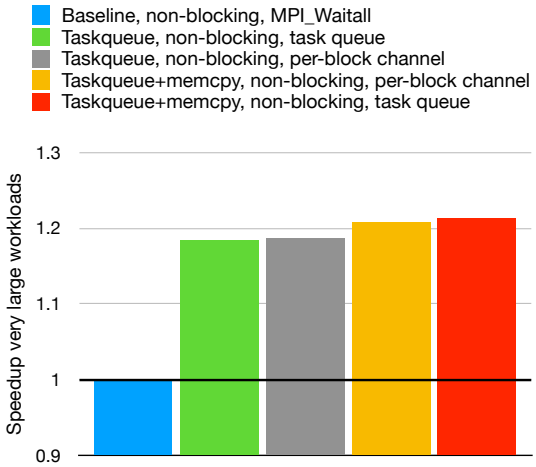


Figure 4.17: *Speedup of selected implementations on NVidia with very large workloads. The baseline is set as reference, and the speedups of taskqueue approaches range from* 1.18 *to* 1.21 *This is the only workload where memcpy outperforms pinned host memory taskqueue.*

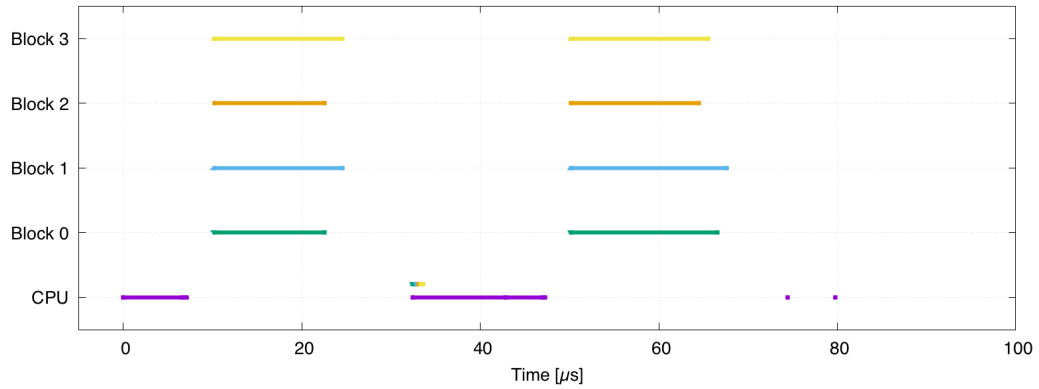
the synchronization points affect the performance.

On the second figure, the taskqueue approach processes the GPU tasks with a visible delay in the range of microseconds. The delay can be seen between a GPU block finishing packing and a dot of the respective colour at the 'CPU' level, which indicates processing of the task by the CPU. As the packing tasks are very small, the delay causes the communication to be processed after the packing finishes. Nevertheless, there are still the following performance gains over the baseline there:

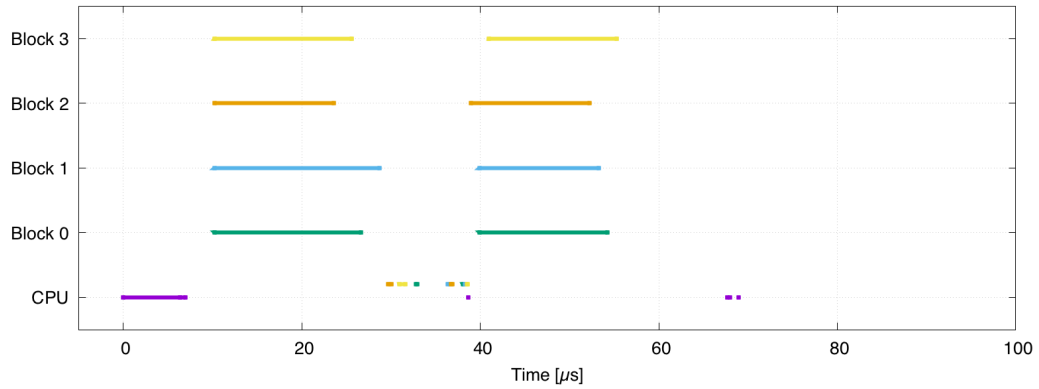
1. There is no delay due to waiting for the kernels to terminate (`hipDeviceSynchronize()`) after packing. That enables triggering the communication sooner compared to the baseline code. Regardless of the approach, some delay is inevitable.

2. There is no overhead caused by launching second kernel for unpacking. The baseline plot indicates finishing the communication phase ca after 43 µs (the dot on the purple 'CPU' line). The rest of the time is spent by launching new kernel. This is not the case for taskqueue plot. There, unpacking starts shortly after the message arrives, which is indicated by the second dot of respective block's colour on the 'CPU' level. There is a small delay there as well, but it is significantly shorter than the one of baseline.

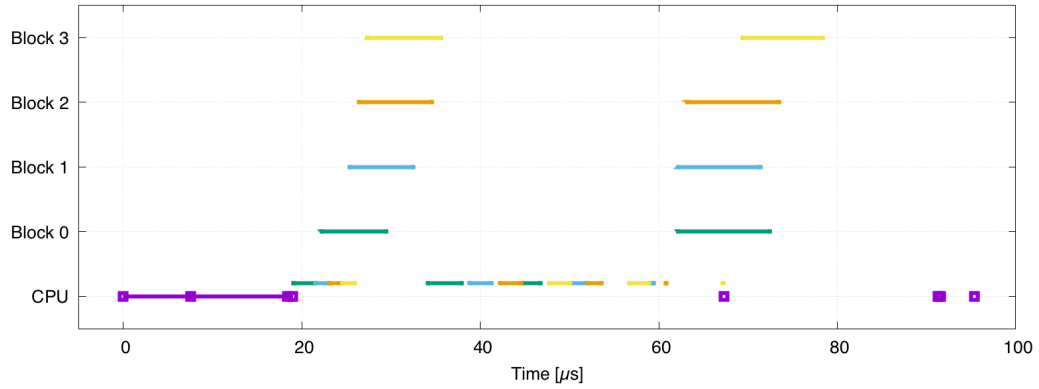
These two observations show that one of the goals of the taskqueue approach – reducing the second kernel launch and termination overhead – is met when using a small workload. However, this observation cannot be generalized to all workloads. As the analyses of previously presented workloads showed, the memory accesses for pinned host memory



(a) *Baseline with non-blocking sends and* `MPI_Waitall`.



(b) *Taskqueue with non-blocking sends and per-block channel.*



(c) *Taksqueue with memcpy, non-blocking sends, and per-block channel.*

Figure 4.18: *One iteration of different implementations with very small workloads on NVidia. Taskqueue with pinned host memory outperforms baseline, memcpy approach has too many overheads for this smaller size.*

implementation constitute the bottleneck of the implementations. The communication between the host and device is implemented using the same channel – pinned host memory. Therefore, the CPU-GPU callbacks can be affected by the tardiness of taskqueue-related write and read operations.

The last plot – 4.18c – focusses on the *memcpy* version. The plot reveals several characteristics of the approach:

1. The kernel launch is followed by a sequence of asynchronous memcpy operations. The duration is ca. 10 µs, which equals to 2.5 µs per asynchronous call. The first set of events denotes the finished *memcpy* operations, and it informs the blocks that they can start packing. Issuing all *memcpy* operations quickly is crucial because until it is done, the CPU cannot move on to processing the events that are already waiting.

2. Due to packing being started as a response to an event, each block starts packing at different time.

3. Processing the events takes a significant amount of time – presumably the `himStreamQuery` call, which checks the status of the *memcpy* operations, is a very expensive call compared to checks for other events. It being called repeatedly for each block makes the test for an event more costly. This characteristic is supposed to be even more significant for workloads with more blocks, where more such calls need to be done.

Very large workloads

The performance comparison using the large set of data is shown in figure 4.17. The taskqueue approach is clearly faster in all tested implementations. For pinned host memory taskqueue, the execution is 18.4 % paster for task queue and 18.7 % for per-block channel variants. The device memory options reach even higher efficiency – task queue is 20.7 % faster, per-block channel 21.3 %.

The very long packing times of pinned host memory versions compared to device memory ones is captured in figure 4.19. There is, again, a clearly visible distinction between the respective variants. This characteristic will likely limit performance of all approaches using pinned host memory combined with large amount of concurrent threads reading and writing the data. Similarly to the previous plot with 27 blocks with 128 threads each (cf. 4.13), the packing times for memcpy approaches are four times longer for the largest buffer – 5.8 −5.9 ms for device memory and 23.2 −23.4 ms for pinned host memory data accesses.

It is also worth mentioning that for the pinned host memory versions, the packing times are very stable across the different implementations, yet they seem very unstable with regards to linearly growing buffer sizes. This characteristic may have something in common with scheduling the GPU blocks on the compute units on the GPU hardware. It is possible that some compute units will have slightly shorter links to the PCIe bus, so that their operations get processed faster compared to other blocks scheduled on another CUs. However, this explanation is only a guess, not a qualified estimation.

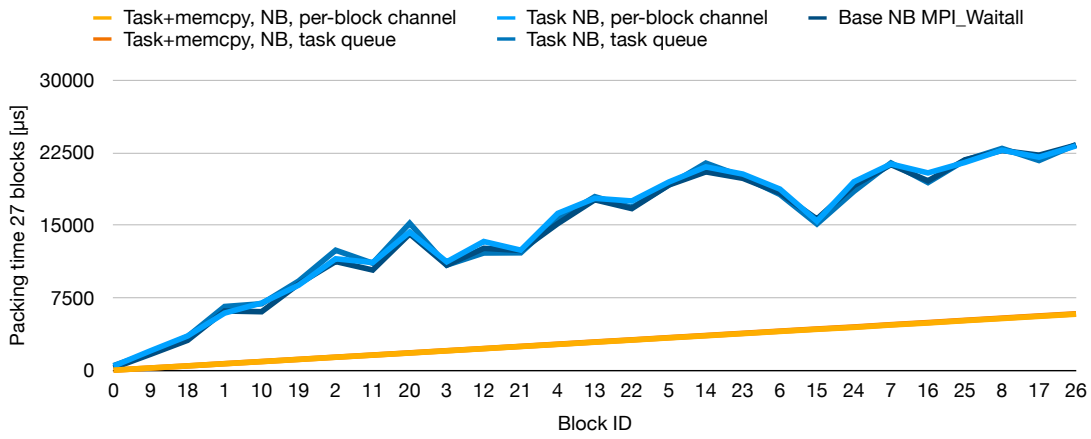


Figure 4.19: *NVidia packing times with very large workload. Orange lines represent taskqueue memcpy options, blue lines baseline and taskqueue pinned host memory options. (Base=baseline, Task=taskqueue, NB=non-blocking sends)*

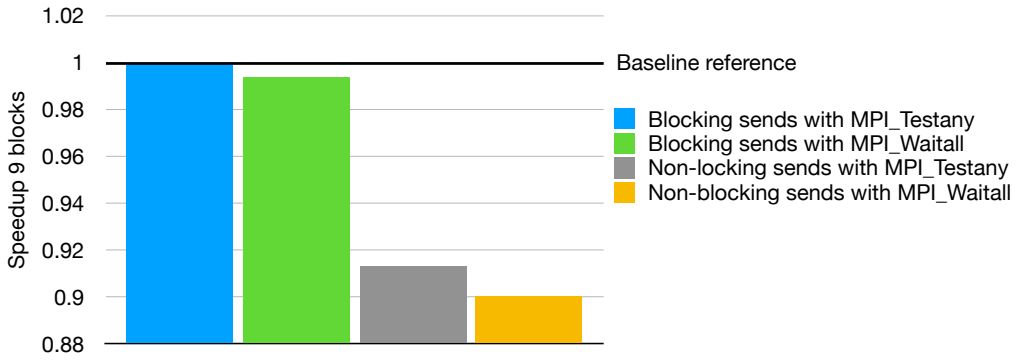


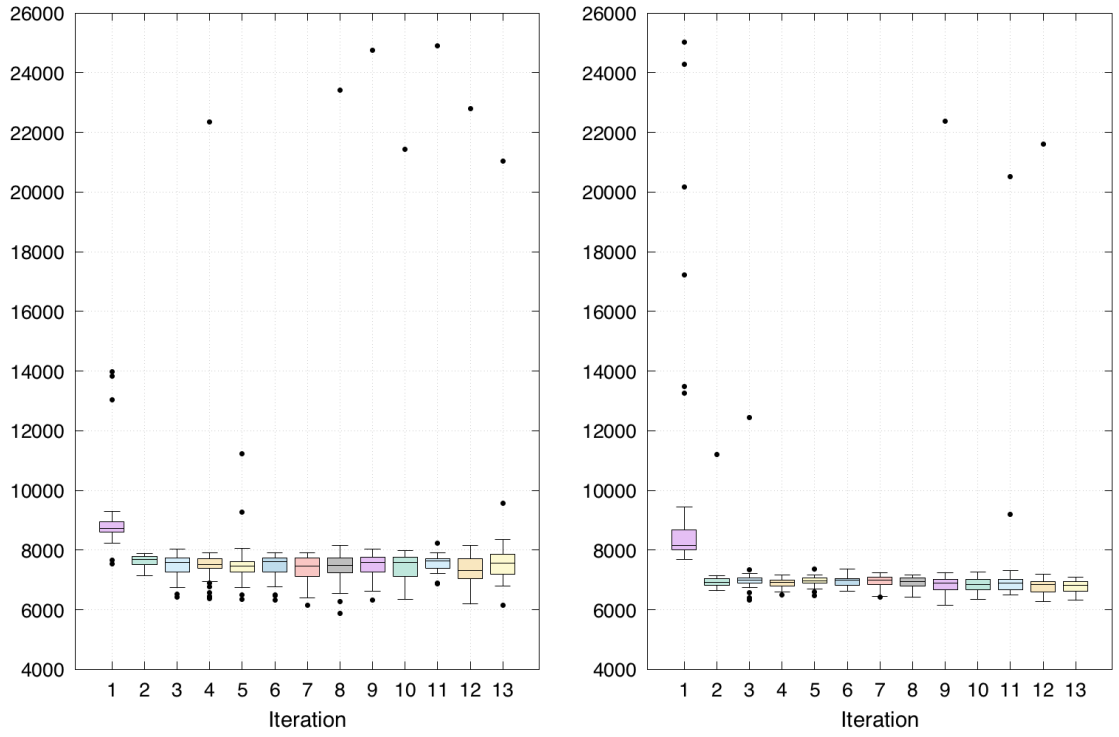
Figure 4.20: *Coherent baseline implementations speedup on AMD with 9 blocks. Blocking communication outperforms the non-blocking one by approx. 9 %.*

4.6 AMD testbed performance

4.6.1 Coherent baseline

The AMD testbed underwent performance tests over the default set of data – the setup utilizes 9 GPU blocks with 128 threads each. The set of measurements was conducted over the same set of implementations as on NVidia testbed, and non-coherent baseline measurements were added. Figure 4.20 shows performance comparison of the four coherent baseline implementations – using MPI blocking or non-blocking communication, and using `MPI_Waitall` or `MPI_Testany`. As the blocking option with `MPI_Testany` is the fastest one, it will be used as a reference. Contrary to the NVidia testbed, blocking options outperform the non-blocking ones by approximately 9 % both using `MPI_Waitall` and `MPI_Testany`. `MPI_Testany` outperformed `MPI_Waitall` in both of these measurements, however only by a very small margin (approximately 1/8 and 1/5 of standard deviation).

The stability of blocking and non-blocking iteration times of `MPI_Waitall` option are



(a) *Baseline coherent non-blocking* `MPI_Waitall`. **(b)** *Baseline coherent blocking* `MPI_Waitall`.

Figure 4.21: *Boxplots showing stability of the iterations on AMD testbed. Y axis presents times per iteration in* µs*. The first iteration takes longer, and is a lot less stable. Other iterations have very few yet significant outliers.*

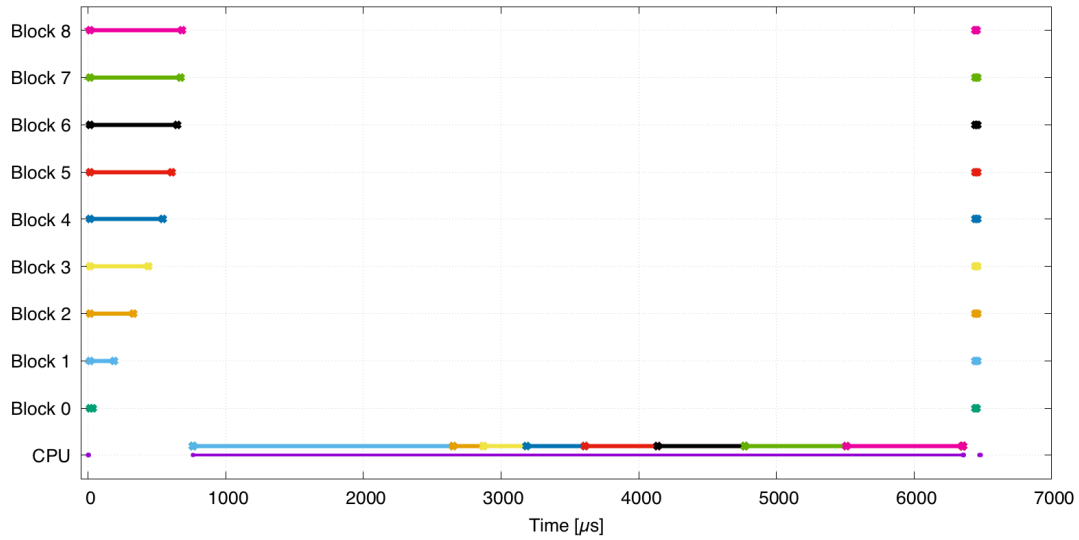


Figure 4.22: *One iteration of blocking baseline with* `MPI_Testany` *on AMD. The 'Block' lines denote packing and unpacking times. The lines of respective colours denote time of blocking send operations for the respective blocks. Block 1's send operation is disproportionally long.*

presented in boxplot in fig. 4.21. The standard deviation over the set of measurements was between 5.6 % and 7.9 % of the iteration time, respectively. These values are not as favourable as ca. 3 % for NVidia testbed. The blocking options outperform its non-blocking counterparts by more than one standard deviation. Thus, this observation can be considered reliable. (Refer to section 4.6.1 for a possible explanation.)

`MPI_Testany` performs slightly better using both blocking and non-blocking communication. However, as the performance difference is only around 1 %, no conclusions can be made based on this observation. Still, it is safe to say that `MPI_Testany` does not present any observable performance penalty on the AMD testbed either.

One baseline iteration

Figure 4.22 captures one iteration of the reference baseline implementation – using MPI blocking communication with `MPI_Testany`. The packing and unpacking operations are presented for each GPU block. The same colour represents the blocking `MPI_Send` operations for each of the blocks. It is displayed on the 'CPU' level as the CPU performs these operations.

The plot is missing a green interval for *block 0* at the beginning of the communication stage. In reality, it is just too short to be observed. The (blocking) operation only takes 460 ns. Contrary to that, *block 1*, even though its buffer is the second smallest, takes the most time to send. The blocking call returns after 1.89 ms, which is more than 4000× slower than *block 0* and ca 8.7× slower than *block 2*, whose buffer is twice as large. On NVidia testbed, such disproportionally long interval on *block 1* was not observed. There, *block 1*'s blocking send is proportional to other blocking send operations.

The reason for the fast return of `MPI_Send` on *block 0* is that its size is less than the

eager limit. This means that the MPI engine can put the data directly on the network (or in this case probably copy the buffer in memory, as both processes reside on a shared memory system) without communicating with the target process. As the data is sent immediately, the blocking call can be returned because the buffer can be reused.[6] Contrary to *block 0*, *block 1*'s buffer is above the *eager limit*, which means that it uses the *rendezvous protocol*. In order for the `MPI_Send` operation to finish, the recipient is first contacted, and has to acknowledge that it is ready to receive. Only then the message can be sent, and the call can return. Other larger blocks use the *rendezvous protocol* as well.

Another experiment was conducted to detect the source of the long sending time for *block 1*. In this experiment, *block 0*'s buffer size was increased to 40 kB (i.e. 1/3 of *block 1*'s buffer size and 1/6 of *block 2*'s buffer size), so that it would be sent with *rendezvous protocol* as well. The sending times were 2175 µs for *block 0*, 107 µs for *block 1*, and 212 µs for *block 2*. This indicates that the first `MPI_Send` operation, presumably to each target MPI rank, in each iteration using *rendezvous protocol* will suffer from this performance penalty. Arguably, some optimizations are in place so that this slowdown only appears in the first case. However, no clear indication is given on why the slowdown reappears in each iteration and not only the first time issuing the communication.

A possible explanation (not verified) is that in the first *rendezvous* send operation, some preparation workload is performed for all *receive buffers* at once. It is possible that the recipient prepares all *receive buffers* and informs the sender already in the first response that all buffers are ready to receive data. As an `MPI_Irecv` operation for each blok is posted at the beginning of each iteration, the receive buffers are already defined for the rest of the send operations. In the next iteration, the same scenario would reoccur for a new set of `MPI_Irecv` operations and their buffers. Should this scenario be indeed really the case, the performance could be improved by sending a "dummy message" using the *rendezvous protocol* right after posting the `MPI_Irecv` operations in each iteration without waiting for the first block to finish packing. This approach was, nevertheless, neither implemented, nor tested. The topic is discussed more in section 4.6.3.

As this phenomenon was neither observed on the NVidia testbed, nor was it present on this testbed's non-coherent tests, it is very likely that it has something to do with AMD's approach to assuring coherence. Presumably, the coherency mechanisms do not work well combined with MPI using the coherently allocated pinned host memory buffers as source and target buffers of send and receive operations.

Full problem size

Figure 4.23 presents comparison of performance over the full problem size – 27 blocks running with 128 threads each. In this setup, the non-blocking option with `MPI_Waitall` is the most efficient one and will set the reference performance for this workload size. As well as in all previous measurements, the performance of `MPI_Waitall` and `MPI_Testany` options is very similar. Contrary to the previous measurements, non-blocking implementations outperformed the blocking ones. The performance of the blocking variants is at 85.6 % of the reference performance for `MPI_Testany` and 85 % for `MPI_Waitall`.

[6] Blocking MPI send calls halt the execution until the send buffer can be reused. When a blocking MPI send operation returns, there is no guarantee that the recipient already has the message.

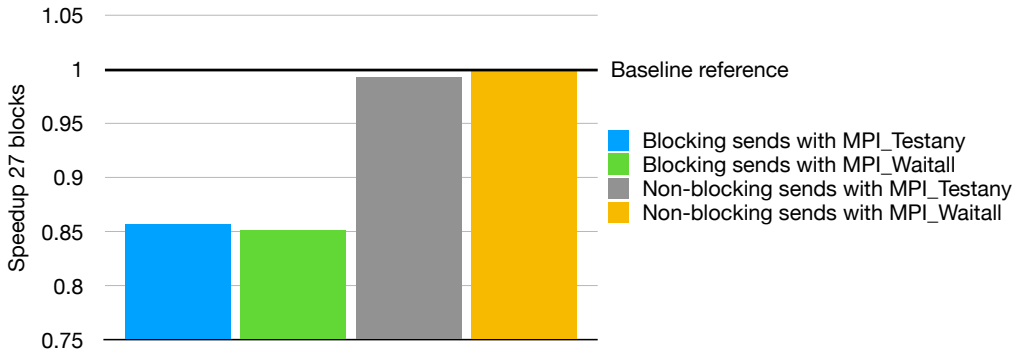


Figure 4.23: *Coherent baseline implementations speedup on AMD with 27 blocks. Unlike in fig. 4.20 with 9 blocks, non-blocking communication outperforms the blocking one. The difference is approx. 15 %.*

In contrast to NVidia baseline tests, where blocking communication was steadily around 3 % slower than non-blocking, the baseline performance changes with workload size on this testbed. Likely, the difference in behaviour on both testbeds is caused by one or both of the following:

- Each testbed uses different version of OpenMPI (AMD runs version 4.0.4, and NVidia 2.1.1). For example, in the more recent version, there may be optimizations that either improve performance of one type of sends, or where a new optimization collides with HIP runtime, which decreases the overall performance as a result.
- Many details can influence MPI performance. The systems are composed of different hardware, not only different GPUs. Also, the low-level software and drivers that control the hardware differ.

Blocking sends operate obliviously to one another by design – before an `MPI_Send` begins, the previous one has already finished. Therefore, a batch of blocking send operations to one target rank, which happens here, can hardly be optimized. This is not the case for `MPI_Isend`. It is safe to assume that the underlying MPI implementation may attempt to optimize the non-blocking sends. They are all posted within a very short time period, and have the same target rank. Arguably, the MPI implementation may merge the buffers into one large message, and send it in one piece to avoid repeating some overheads. Then, the recipient would split the message back to the original pieces.

Assuming this, the reason for different performance may be that the overheads to perform the optimization for 8 blocks[7] come with greater performance penalty than the actual performance gain is. In case the performance penalty is constant, the performance gain for 26 blocks may already prevail and yield a speedup.

Nevertheless, this is only a weak argument especially because the performance differences were too large. Therefore, this issue shall be investigated more in detail in order to provide a solid argument.

[7]The first block is sent eagerly and is likely already on the network by the time the last `MPI_Isend` is issued.

Taskqueue performance results – AMD

	Task queue	Per-block channel
Using pinned host memory:		
MPI non-blocking sends (speedup vs. non-blocking)	3539 ms (0.002)	3563 ms (0.002)
MPI blocking sends (speedup vs. non-blocking)	1733 ms (0.004)	1671 ms (0.004)
Using device memory + memcpy:		
MPI non-blocking sends (speedup vs. non-blocking)	6351 ms (0.001)	6167 ms (0.001)
MPI blocking sends (speedup vs. non-blocking)	5645 ms (0.001)	5236 ms (0.001)

Table 4.6: *Taskqueue implementations speedup on AMD compared to blocking and non-blocking references. The presented data address the default workload with 9 blocks. The performance is overall very bad.*

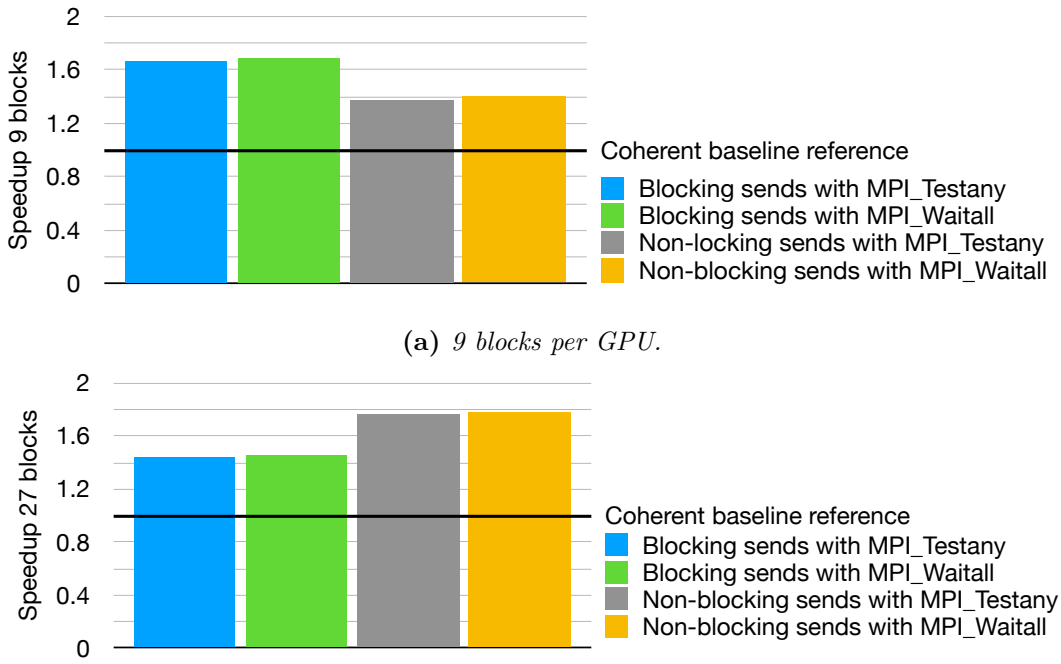
4.6.2 Taskqueue

None of the taskqueue approaches reached any reasonable performance results on this testbed. The tests always finished, and a wrong value was never observed, so there is no reason to suspect incorrect bahaviour. Nevertheless, the performance is very poor; not even near the baseline. It is slower by two to three orders of magnitude.

The problem on this testbed lies in slow propagation of the CPU-to-GPU and GPU-to-CPU communication. The delay of the communication between the parties varies a lot. For majority of the cases, the callbacks work as expected in both directions, but sometimes, a callback takes extremely long time to propagate. As the callbacks are an essential part of this algorithm, there is no easy way to circumvent the problem.

This very slow propagation may be caused by writing to and ceaselessly reading from one piece of memory. For example in the task queue approach, the CPU repeatedly reads the front of the task queue. At the same time, the GPU attempts writing into this memory location as well. In the oppiosite direction, GPU blocks spin-wait on an array index, where the CPU writes upon receiving an MPI message. This problem has very likely already manifested itself in the `hipMemcpyAsync` deadlocks described in section 3.7.2. In order to find out details about how concurrent writes from one party and reads from the other work, more detailed research would have to be conducted.

Still, there are some trends in the performance results (presented in table 4.6) that can be observed, and they support the statement made previously. The blocking versions, especially using pinned host memory, outperform the non-blocking ones. This is presumably due to the fact that while the send operation blocks the execution, there are no reads of the critical memory locations from the CPU. In this time interval, it might be easier for the GPU to write there. Similar speedup was achieved by putting a *sleep* command at the beginning of each check for events in non-blocking version. Additionally, the pinned host options perform better than the device memory ones. This is seemingly due to the fact



(a) *9 blocks per GPU.*

(b) *27 blocks per GPU.*

Figure 4.24: *Non-coherent baseline implementations speedup on AMD with 9 and 27 blocks. Analogously to fig 4.20 and 4.23, the blocking options perform better on the smaller workload and the non-blocking ones on the larger workload. The difference is even higher – ca. 20 and 22 %.*

that the *memcpy* implementations contain additional CPU-GPU callbacks that manage the memory copies.

4.6.3 Non-coherent buffers in baseline

Figures 4.24a and 4.24b show the performance of baseline code with non-coherently allocated *send* and *receive buffers* using 9 and 27 blocks, respectively. The first case yields a speedup of 1.68 and the second one of 1.77 These performance gains are significant. None of the taskqueue approaches yielded any performance gain on this platform, so this was the only, yet rather simple optimization to the baseline code that managed to improve its performance.

Analogously to the coherent options, the smaller workload favours the MPI blocking implementations, while the larger workload favours the non-blocking communication. For non-coherent options and the default 9-blocks workload, the blocking communication performs 20 % better than non-blocking. For the full problem size workload with 27 blocks, the non-blocking communication outperforms blocking options by 22 %. Yet even the worst-performing non-coherent option clearly outperforms any coherent option.

There are two reasons for the better performance of non-coherent memory:

1. The packing times are shorter in general, therefore the MPI communication phase

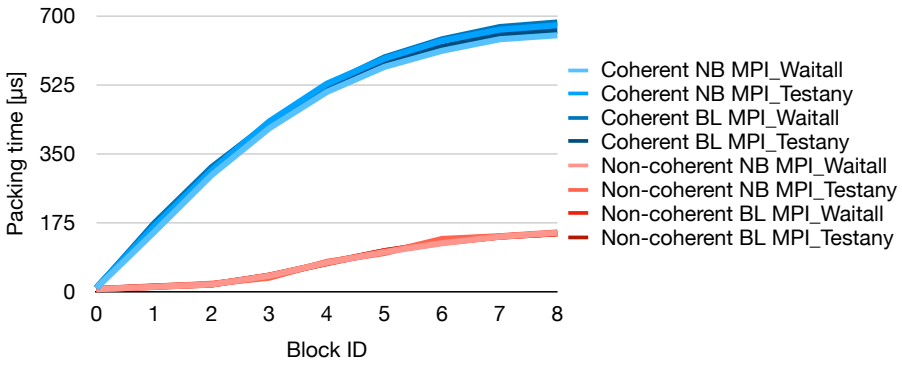


Figure 4.25: *AMD baseline packing times with 9 blocks. Red lines represent non-coherent memory options, blue lines the coherent ones. (BL=blocking sends, NB=non-blocking sends)*

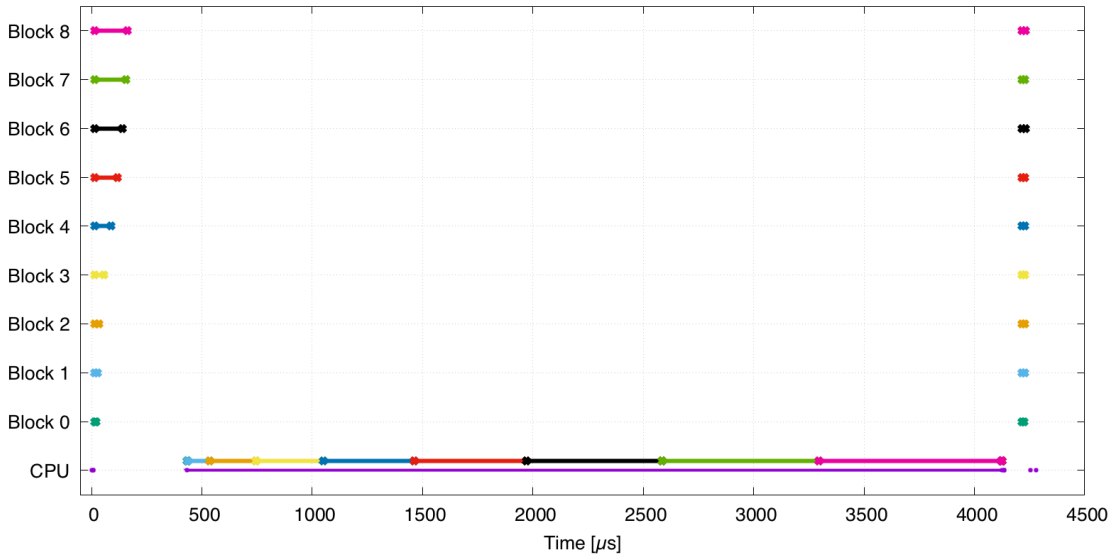


Figure 4.26: *One iteration of non-coherent blocking baseline with* `MPI_Testany` *on AMD. The 'Block' lines denote packing and unpacking times. The lines of respective colours denote time of blocking send operations for the respective blocks. Unlike in fig. 4.22 with coherent buffers, Block 1's send operation is proportional to others.*

can start earlier. Figure 4.25 compares packing times of baseline implementations using coherent and non-coherent memory. The blue (slower) lines use coherent option. The largest buffer takes on average between 650 µs and 680 µs to pack. Contrary to that, the lower lines, that are hard to even distinguish from one another, represent the non-coherent implementations. They only need between 148 µs and 150 µs to pack all messages, which is more than four times faster.

2. As the packing takes only a small portion of the iteration time, and most of the time is needed for MPI communication, even such a drastic reduction of packing time would not be enough to justify the speedup. Figure 4.26 visualises one iteration of non-coherent blocking baseline with `MPI_Testany` (cf. coherent option in fig. 4.22). This figure clearly misses the long delay in the first blocking send operation using the *rendezvous protocol*. This difference is even more significant than the shorter packing times, and is therefore responsible for the most of the speedup.

Figures 4.26 and 4.22 show that the coherent or non-coherent buffer allocation type affects the performance of MPI communication. Allocating coherent pinned host memory brings a performance penalty to the first *rendezvous* MPI Send operation, which is not present in the non-coherent case. A possible yet not verified explanation is that the coherent allocation uses some synchronization mechanisms to ensure coherency of the data. Those mechanisms would be triggered to ensure coherency before MPI is allowed to write to the buffers. As there is no kernel operating over the set of *send* and *receive buffers* at that time, any synchronization operation would be redundant. However, this fact may be unknown to the HIP runtime, so the synchronization would be performed anyway. The coherency and synchronization mechanisms may be the source of the long delay before the first message can be sent or received. It is more likely that the issue would be present on the receiver side, because it occurs only in the first blocking send operation, however any eventuality is possible. As the non-coherent allocation does not provide coherency, no such mechanisms need to be in place, thus no tardiness occurs.

The functionality of the coherent and non-coherent memory, and its interactions with MPI communication should be further examined. As the non-coherent memory implementation brought a large speedup, and it is described in a very general manner in the HIP documentation, knowing more about the behaviour would be very beneficial.

Coming back to figure 4.25, it is worth noticing that the coherent approaches' curve is rather concave than a straight line. The buffer size, however, increases linearly. As all blocks start computing at the same time, the number of concurrent threads writing to a *send buffer* decreases each time a block finishes packing. There is again a possibility of congesting the host memory or the PCIe bus with too many parallel accesses, as it was the case in NVidia tests. As the number of concurrent threads decreases in time, the congestion would be reduced and the write latency would decrease. Hence, this offers an explanation of the concave curve. In case this speculation is correct, the AMD testbed pinned host memory access pattern with congested system is more fair than the one of NVidia testbed.

Finally, figure 4.27 presents a boxplot that shows the average times and stability of all

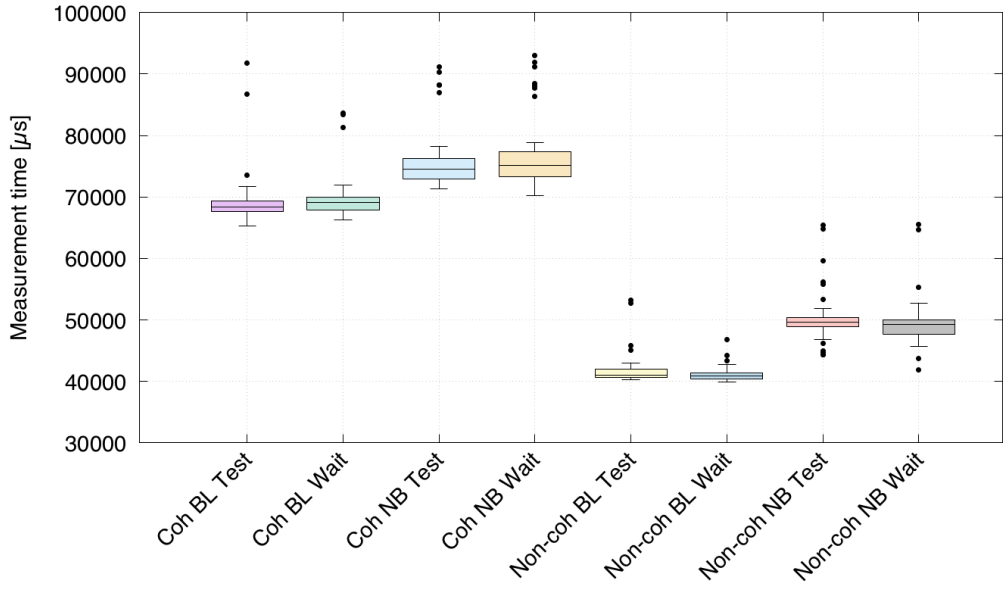


Figure 4.27: *Boxplot with summary over measured times of baseline coherent and non-coherent experiments on AMD with 9 blocks. The non-blocking versions have more outliers and the stability of the results is lower, but overall, the results are reliable. (Coh=coherent, Non-coh=non-coherent, BL=blocking sends, NB=non-blocking sends, Test=`MPI_Testany`, Wait=Test=`MPI_Waitall`)*

baseline versions. The taskqueue approach was not plotted because it would require a completely different scale of the plot.

4.7 Summary of the performance measurements

Probably the most interesting finding from the performance measurements is that both testbeds demonstrated a very different behaviour. Even though they compiled and ran the same code, the system behaviour and the performance results were very different.

On the NVidia testbed, the performance results came out as expected. The pinned host memory taskqueue approach outperforms baseline in all cases. *Memcpy* variant does not perform well for very small workloads, but in all the other test cases, it outperforms the baseline as well.

Figure 4.28 visualises how the achieved speedups evolve with increasing workload. The workload size increases from left to right on the x axis. The *memcpy* versions (lines in shades of red) achieve higher speedup with larger workloads, whereas pinned host memory taskqueue (lines in shades of blue) achieves a more constant speedup even for small workloads. Based on the set of performed tests, the best approach for smaller workloads is pinned host memory taskqueue. For larger ones, *memcpy* is similarly performant and eventually becomes superior.

The next observation was that the implementation without atomic operations (per-block channel) slightly outperforms the implementation with them (task queue) in the vast

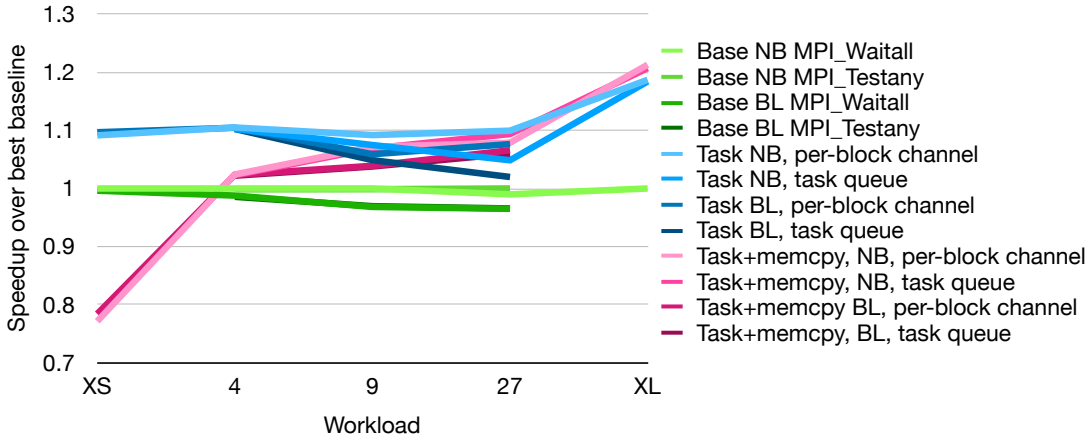


Figure 4.28: *Speedup over all workloads on NVidia. The workload size is ascending. Green lines represent baseline, blue ones taskqueue with pinned host memory, and purple ones taskqueue with memcpy. (Base=baseline, Task=taskqueue, BL=blocking sends, NB=non-blocking sends)*

majority of cases. This observation indicates that the extra work done by the CPU, i.e. checking a separate index for each block, is less of an overhead than using atomic operations to write in the single task queue.

The main bottleneck of the pinned host memory variants is the congestion when many parallel threads read from and write to the buffers. One of the bottlenecks of *memcpy* options is likely the single thread on the CPU side. It does not manage to process the large amount of events swiftly. `hipStreamQuery` call turns to be a very expensive and slow call.

The AMD testbed did not yield expected results. The taskqueue approach has revealed not to be a suitable solution for this testbed. The source of the problem are CPU-to-GPU and GPU-to-CPU callbacks that sometimes take extremely long time to propagate. In the background, this problem probably translates to writing to and reading from the same memory location by the CPU and the GPU simultaneously.

Moreover, the performance of blocking and non-blocking baseline options differs with workload size – for the smaller workload, blocking options perform significantly better, while for the larger workload, the performance of the non-blocking option is substantially better.

Finally, using the non-coherent pinned host memory has shown to be a more efficient solution. The speedup over the coherent option is significant. Nevertheless, a large part of the worse performance of the coherent option is caused by its much longer than expected first *rendezvous* send operation in each iteration. As this is neither the case for the NVidia testbed, nor for the non-coherent buffers, some sort of suboptimal cooperation between the coherent buffers (and possibly some coherency-related mechanisms) and the MPI runtime using these buffers might be causing this problem.

5 Summary and outlook

This work focussed on data transfers between GPUs in potentially multi-node distributed memory systems. The scenario for the communication was halo exchange. Halo exchange is typically the last step of a distributed stencil computation, where border cells ('halo layer') are exchanged between two adjacent components of a grid. The data for the exchange are put together by GPUs, and on the destination node, a GPU distributes the contiguous data again. Stencil codes usually consist of a large number of such iterations. The use-case was defined with 27 blocks per GPU, whose halo layers range from 1 B to 1 MB and the communication scheme is static and known at the beginning of the program.

The goal of this work was to design, develop, and test an efficient solution for a quick transfer of data in such scenario. The solution should be platform-agnostic, i.e. it should run on both AMD and NVidia GPUs. Having both an AMD and an NVidia testbeds available, another task was to evaluate performance of proposed algorithms, and to observe the differences between both platforms.

5.1 Implemented solutions

Two main approaches were developed – baseline with a rather naive sequential workflow, and taskqueue with a more dynamic workflow that aims at overlapping computation and communication.

The baseline solution workflow is divided into separate phases that do not overlap. In the first stage, the GPU prepares the buffers to send (packing). Each GPU block fills one buffer independently on each other. This phase ends when all buffers are prepared and the data is available to the CPU. Then, the MPI communication stage takes place. The CPUs exchange the buffers. When a CPU received all buffers, the GPU starts to process the received buffers (unpacking). When all MPI processes are done, the iteration finishes.

The taskqueue approach was designed as a response to analysis of the baseline workflow, and the goal was to eliminate the identified flaws of that workflow. The two improvements were:

- Send a message via MPI as soon as the buffer is available.

  There are different buffers for each GPU block. Some blocks finish preparing the *send buffer* faster than others. These buffers should be sent immediately over MPI to the target, not first after all other GPU blocks have finished the work as well.

- Eliminate kernel launch and termination (synchronization) overhead of the second kernel in each iteration. Hence, there should be one kernel that lasts throughout the whole iteration.

Both goals were achieved by designing a structure that enables communication between the CPU and GPU. It is done through pinned host memory, which both parties can access. This structure enables the workflow to be more dynamic. The CPU takes the slave role and dynamically responds to events that occur in the system. The events include the GPU blocks finishing preparation of the *send buffer*, completed MPI message transfers, and in some versions the completion of *memcpy* operations between the host and device.

Each solution came in different flavours. If there were multiple implementation decisions possible, several options were implemented in order to assess the effect of a particular decision. Baseline code had three such options:

1. Using MPI blocking and non-blocking communication for send operations.
2. Using `MPI_Waitall` and a loop of `MPI_Testany` calls to halt execution until the communication is done.
3. Allocating buffers coherently and non-coherently. The latter is only available on AMD devices.

Taskqueue also had three such options:

1. Using MPI blocking and non-blocking communication for send operations.
2. Using atomic operations on the GPU side to write into a task queue shared among GPU blocks, or an implementation where each GPU block writes to a different address, so no atomics are necessary.
3. The GPUs can either use pinned host memory buffers for preparation of the message, or device memory buffers are used and copied between the host and device.

5.2 Performance results

In summary, the performance results were as expected on the NVidia testbed, where a speedup was achieved, while the AMD testbed yielded some unexpected results.

5.2.1 NVidia

On the NVidia side, the taskqueue approach outperformed the baseline code. The performance gain differs with a changing workload, but in every test case, the best performing taskqueue version outperforms the best performing baseline version by 9 % to 22 %. Comparing the implementation options, the following observations were made:

- Apart from the test case with very small buffer sizes that are sent with the *eager protocol*, the non-blocking communication options outperform their blocking counterparts by roughly 3 %.
- The performance of `MPI_Testany` and `MPI_Waitall` options in the baseline code is identical, therefore, there is no apparent performance penalty for using `MPI_Testany` in the taskqueue code.

- The taskqueue implementation without atomic operations (per-block channel) outperforms the versions with atomics. It can be concluded that the GPU-side overhead for using atomic operations to write into the common task queue is higher than the CPU-side overhead to check one different memory location for each of the blocks.
- Especially for smaller workloads, the pinned host memory option performs better than device memory with *memcpy*. Nevertheless, as the workload size grows, the performance of the latter increases until it eventually becomes the faster option for the largest test case. Contrary to that, speedup is relatively stable for the former option.

There were multiple bottlenecks identified that hinder the performance. When too many GPU threads write to pinned host memory buffers, the system gets congested and write times increase. With a small number of parallel threads, pinned host and device memory packing times are almost identical. When the number of parallel threads increases, the pinned host option packing times are several times longer than the *memcpy* ones. The second bottleneck is that, especially for *memcpy* approach with more events, the single-threaded CPU event processing mechanism was sometimes not able to process all events instantly. Finally, the callbacks have the form of writing to and reading from pinned host memory. When the memory writes and reads are congested because of the packing workload, the callbacks are also affected by the delays.

5.2.2 AMD

The performance of the taskqueue approach on the AMD testbed was poor and the taskqueue did not prove to be a suitable solution for this testbed. The main problem of the approach on this platform lies in extreme delays that occur while communicating between the parties.

Concerning the baseline approach, the following observations were made:

- For the smaller of the two measured workloads, blocking communication outperformed non-blocking one in all tested versions. For a larger workload, the performance was exactly the opposite. The differences were very large in both cases. With the smaller workload, the fastest coherent blocking version was 9.5 % faster than fastest coherent non-blocking version, and the fastest non-coherent blocking version was 20.5 % faster than the non-coherent non-blocking one. On the contrary, with the larger workload, non-blocking coherent version was 16.8 % faster than the blocking one and for the non-coherent options, this difference was 21.8 %.
- The performance of `MPI_Testany` and `MPI_Waitall` options in the baseline code is very similar as it was the case for the NVidia testbed.

Measuring the performance of the non-coherent baseline option, a large speedup of up to 1.77 was measured. The speedup is much larger than the one achieved with taskqueue approach on NVidia. The concept of non-coherent memory is not compatible with the taskqueue approach logic, so even if taskqueue approach had achieved good performance results on the AMD testbed, the concepts could not be easily combined.

It is also worth mentioning that the speedup of the non-coherent option may be caused rather by poor performance of the coherent implementation than a good performance of the non-coherent one. The coherent option suffered from delays in MPI communication that were neither present on (coherent) NVidia testbed, nor using the non-coherent implementation.

5.3 Conclusion

In general, the cooperation of GPUs and MPI may be problematic. Combining the two concepts is often not straightforward. For NVidia devices and CUDA, some MPI implementations offer CUDA-aware functionality that targets these use-cases, however, the solutions are usually platform-specific, may require special hardware, and are not part of every MPI implementation. These reasons discarded CUDA-aware MPI from our focus.

The proposed solution – the taskqueue approach – yielded solid performance gains on NVidia platform and can therefore be regarded as a suitable solution on this platform. Unfortunately, the results on AMD were not satisfactory, and thus the proposed solution is not platform-agnostic, which was one of the goals. Based on the observations made during the performance measurements, there is most likely no single universal solution that would yield a close to maximum performance on both testbeds – while for NVidia testbed, the taskqueue seems to be the right approach, the best solution on AMD testbed will very likely deal with non-coherent pinned host memory. The current form of the taskqueue algorithm is not compatible with non-coherent memory, so the resulting solution would probably be a trade-off for both platforms.

On the other hand, the discrepancy in the performance results discovered many differences between the two platforms, which is a valuable discovery, and it was also one of the goals of this thesis. Reasons for the unexpected behaviour were always offered, however they need to be examined more closely and confirmed. Being aware of the differences, and ways to overcome them, it should be easier in the future to develop and tune performance of applications made for both platforms. The findings can also solve as a basis for the vendors to improve performance of their products, and to align the characteristics of their products in order to arrange easier and more efficient development of platform-agnostic GPU applications.

HIP is a promising project that may enable easy transformation from vendor-locked CUDA codes to portable codes. Its syntax is very similar to the one of CUDA, so the learning curve for CUDA developers is very steep. Nevertheless, after having worked with HIP for several months, there were several deficiencies identified. The most limiting aspect is the lack of a comprehensive documentation. At the time of implementing the solution, CUDA documentation was often the only source of documentation and specifications even for development on AMD. Compared to CUDA and NVidia, working with HIP on AMD is currently a lot less pleasant experience because of the insufficient documentation and unexpected behaviour and bugs that were encountered during the implementation phase. Comparing the two platforms (AMD and NVidia) and their ecosystems, AMD with HIP makes an impression of an inferior solution to work with that is not yet mature enough, and is so far not well suited for uncommon use-cases.

5.4 Outlook

The designed algorithms, findings, and discoveries from the performance evaluation give rise to two types of possible follow-up work – one that would more closely investigate some observed phenomena and potentially prove or disprove the offered explanations, and another one that would improve the solution design by mitigating the discovered bottlenecks.

Chapter 4 discussed several observations that were not expected and that should be studied in order to understand what is happening in the system. Therefore, there are possible research directions that would uncover the system behaviour.

- On the AMD testbed, very large differences between MPI blocking and non-blocking communication were observed. A subsequent research that would comprehend and describe the reasons for this behaviour, and why it is not so on the NVidia testbed, would help understanding the differences between the systems. The research should also clarify which role do HIP pinned host memory buffers play. An expected output would be a framework that finds the workload limit that makes one or the other communication type more efficient. Moreover, if some changes to HIP or OpenMPI library would improve the behaviour, they could be identified.

- On the NVidia testbed, the pinned host memory packing functions suffered from poor behaviour with increased workload. First of all, the system component causing this congestion has to be identified. It is worth researching how the data is transferred between the main memory and the GPU memory, and if there is anything that could improve the performance.

- There is very little description provided of the HIP coherent and non-coherent pinned host memory types. As the non-coherent option presented a significant speedup to the coherent baseline solution on AMD, it is important to understand why it is so. First, it is necessary to identify the bottlenecks of the coherent versions, and why the same problems do not occur in the non-coherent versions or on the NVidia tesbed. In order to achieve this, a general understanding of the underlying functionality and implementation of coherent and non-coherent memory is necessary. Then, the impact of using these buffers in MPI communication should be understood. In our use-case, using non-coherent buffers reduced the latency of the first *rendezvous* MPI message drastically. Therefore, the overheads of using coherent pinned host memory as MPI buffers on AMD needs to be understood as well.

- Finally, the reasons for poor performance of the CPU-to-GPU callbacks on AMD should be identified as they constitute yet another difference between the platforms.

The performance data collected in chapter 4 showed that the taskqueue approach still might have some room for improvements on the NVidia side, and that it did not work well on the AMD side. Therefore, several design changes can be considered to increase performance on the NVidia side, and to create a new solution that would hopefully bring a reasonable speedup on both sides, even if the speedup is not the best possible:

- At the moment, there is one kernel that handles all blocks. An alternative solution consists in having a series of separate kernel launches that only launch one block. This design would allow synchronization at kernel boundaries on a block granularity, which would be compliant with the non-coherent pinned host memory and would also enable computation and communication overlapping. This algorithm would work on a remotely similar principle to the taskqueue approach, but there would be no CPU-GPU communication over the pinned host memory. Instead, the kernel would terminate, and the host would repeatedly check for terminated kernels and synchronize only them. When the kernel is terminated, the communication could start. Upon receiving a message an unpacking kernel for given block would be launched.

  This concept would have many overheads and delays related to a plethora of kernel launches in each iteration, but should be compliant with the non-coherent memory option, and does not use CPU-GPU callbacks. Therefore, it eliminates both problematic aspects of the AMD testbed. Creating a detailed design of this concept, implementing it, and measuring the performance would reveal the potential of the approach that should be universal.

- The current taskqueue approach (or the new concept proposed above) could be made multi-threaded on the CPU side. The profiling data showed that especially for some variants, the CPU is not idle in critical phases of the runtime, therefore the performance may be increased by introducing multiple threads that concurrently process the events. See section 3.6.1 for a more detailed description.

- The current solution relies on different blocks being packed differently long. As soon as this ceases to be the case, no computation and communication overlapping is to be expected. Pipelining the packing (, *memcpy*) and send operations could introduce overlapping in such use-cases as well. Refer to section 3.6.2 for more details.

- All current pinned host memory implementations suffered from too many concurrent read and write accesses to the pinned host memory. If a mechanism is in place that allows only a certain number of blocks to work at a time, performance might be improved. One can research how such a mechanism would work and how it could be efficiently implemented. If there is an efficient mechanism in place, a subset of blocks could be packed faster, and pushed on the network sooner. Moreover, the CPU-GPU communication, which is affected by the congestion as well, might therefore also work faster.

- Another option is researching on possibilities how to prioritize some memory operations (CPU-GPU callbacks) over others (packing traffic) in a congested system.

List of Figures

List of Tables

Listings

Bibliography

[1] Top500. https://www.top500.org/. Accessed: 2020-08-17.

[2] Advanced Micro Devices, Inc. Hip programming guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/hip-programming.htm, 2020. Accessed: 2020-08-27.

[3] Advanced Micro Devices, Inc. Rocm documentation. https://rocmdocs.amd.com/en/latest/, 2020. Accessed: 2020-08-27.

[4] Advanced Micro Devices Inc. Rocm: Hpc. https://www.amd.com/en/graphics/servers-solutions-rocm-hpc, 2020. Accessed: 2020-08-30.

[5] AMD. *"Vega" 7nm Instruction Set Architecture, Reference Guide.*

[6] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.

[7] Damon McDougall, Chip Freitag, Joe Greathouse, Nicholas Malaya, Noah Wolfe, Noel Chalmers, Scott Moe, Rene van Oostrum, Nick Curtis. Webinar: Introduction to amd gpu programming with hip.

[8] Feras Daoud, Amir Watad, and Mark Silberstein. 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*, pages 1–8, 2016.

[9] Davide Rossetti, SW Compute Team. Gpudirect: Integrating the gpu with a network interface. Accessed: 2020-09-03.

[10] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[11] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[12] Message Passing Interface Forum. Mpi: A message-passing interface standard version 3.1. Technical report, USA, 2015.

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

[14] T. K. Hazra. Parallel computing – defining what it is and the doors it opens via transputers. *IEEE Potentials*, 14(3):17–20, 1995.

[15] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. Impacc: a tightly integrated mpi+ openacc framework exploiting shared memory parallelism. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 189–201, 2016.

[16] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[17] Jiri Kraus. An introduction to cuda-aware mpi. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/, Mar 2013. Accessed: 2020-09-03.

[18] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. Gpu triggered networking for intra-kernel communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[19] Piotr Luszczek. Cuda's asynchronous and non-blocking techniques. Accessed: 2020-08-30, 2000.

[20] Alexander Margolin and Amnon Barak. Rdma-based library for collective operations in mpi. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, pages 39–46. IEEE, 2019.

[21] David L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, IETF Tools, March 1992.

[22] Takefumi Miyoshi, Hidetsugu Irie, Keigo Shima, Hiroki Honda, Masaaki Kondo, and Tsutomu Yoshinaga. Flat: a gpu programming framework to provide embedded mpi. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 20–29, 2012.

[23] NVIDIA. Cuda c++ programming guide. Technical report, USA, 2020.

[24] Jon Peddie. Jon peddie research releases its q3, 2018 add-in board report. https://www.jonpeddie.com/press-releases/jon-peddie-research-releases-its-q3-2018-add-in-board-report, Nov 2018. Accessed: 2020-08-29.

[25] Sushil Prasad, Anshul Gupta, Arnold Rosenberg, Alan Sussman, and Charles Weems. *Topics in Parallel and Distributed Computing*. Springer, 2015.

[26] Rene van Oostrum, Noel Chalmers, Damon McDougall, Paul Bauman, Nicholas Curtis, Nicholas Malaya, Noah Wolfe. Amd gpu hardware basics. Accessed: 2020-08-17.

[27] ROCm Developer Tools. Cuda runtime api functions supported by hip. https://github.com/ROCm-Developer-Tools/HIP/blob/rocm-3.7.0/docs/markdown/CUDA_Runtime_API_functions_supported_by_HIP.md, 2020. Accessed: 2020-08-26.

[28] L Rota, M Vogelgesang, LE Ardila Perez, M Caselle, S Chilingaryan, T Dritschler, N Zilio, A Kopmann, M Balzer, and M Weber. A high-throughput readout architecture based on pci-express gen3 and directgma technology. *Journal of Instrumentation*, 11(02):P02007, 2016.

[29] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R Gregg Brickner. Compiling stencils in high performance fortran. 1997.

[30] Karl Rupp. 42 years of microprocessor trend data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, Feb 2018. Accessed: 2020-08-17.

[31] Kittisak Sajjapongse, Ruidong Gu, and Michela Becchi. Ivm: a task-based shared memory programming model and runtime system to enable uniform access to cpu-gpu clusters. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 154–163, 2016.

[32] Rong Shi, Xiaoyi Lu, Sreeram Potluri, Khaled Hamidouche, Jie Zhang, and Dhabaleswar K Panda. Hand: A hybrid approach to accelerate non-contiguous data movement using mpi datatypes on gpu clusters. In *2014 43rd International Conference on Parallel Processing*, pages 221–230. IEEE, 2014.

[33] Jeff A Stuart, Michael Cox, and John D Owens. Gpu-to-cpu callbacks. In *European Conference on Parallel Processing*, pages 365–372. Springer, 2010.

[34] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing cpu and gpu design trends with product data. *arXiv preprint arXiv:1911.11313*, 2019.

[35] Sun, Yifan and Mukherjee, Saoni and Baruah, Trinayan and Dong, Shi and Gutierrez, Julian and Mohan, Prannoy and Kaeli, David. Evaluating performance tradeoffs on the radeon open compute platform. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 209–218. IEEE, 2018.

[36] Maciej Szpindler. Scalable remote memory access halo exchange with reduced synchronization cost. In *Proceedings of Cray User Group (CUG) Conference*, 2016.

[37] ROCm Developer Tools. Hip. https://github.com/ROCm-Developer-Tools/HIP, 2020. Accessed: 2020-08-26.

[38] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel Loh, Mark Oskin, and Steven K Reinhardt. Gpu system calls. *arXiv preprint arXiv:1705.06965*, 2017.

[39] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K Panda. Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2. In *2011 IEEE International Conference on Cluster Computing*, pages 308–316. IEEE, 2011.

[40] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science-Research and Development*, 26(3-4):257, 2011.