KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree project in Computer Science

Second cycle, 30 Credits

Using GPU-aware message passing to accelerate high-fidelity fluid simulations

JACOB WAHLGREN

Stockholm, Sweden, 2022

Using GPU-aware message passing to accelerate high-fidelity fluid simulations

JACOB WAHLGREN

Master's Programme, Computer Science, 120 credits
Date: June 3, 2022

Supervisors: Niclas Jansson, Martin Karp
Examiner: Stefano Markidis
School of Electrical Engineering and Computer Science
Swedish title: Användning av grafikprocessormedveten meddelandeförmedling för att accelerera nogranna strömningsmekaniska datorsimuleringar

Abstract

Motivated by the end of Moore's law, graphics processing units (GPUs) are replacing general-purpose processors as the main source of computational power in emerging supercomputing architectures. A challenge in systems with GPU accelerators is the cost of transferring data between the host memory and the GPU device memory. On supercomputers, the standard for communication between compute nodes is called Message Passing Interface (MPI). Recently, many MPI implementations support using GPU device memory directly as communication buffers, known as GPU-aware MPI.

One of the most computationally demanding applications on supercomputers is high-fidelity simulations of turbulent fluid flow. Improved performance in high-fidelity fluid simulations can enable cases that are intractable today, such as a complete aircraft in flight.

In this thesis, we compare the MPI performance with host memory and GPU device memory, and demonstrate how GPU-aware MPI can be used to accelerate high-fidelity incompressible fluid simulations in the spectral element code Neko. On a test system with NVIDIA A100 GPUs, we find that MPI performance is similar using host memory and device memory, except for intra-node messages in the range of 1-64 KB which is significantly slower using device memory, and above 1 MB which is faster using device memory. We also find that the performance of high-fidelity simulations in Neko can be improved by up to 2.59 times by using GPU-aware MPI in the gather–scatter operation, which avoids several transfers between host and device memory.

Keywords

high-performance computing, computational fluid dynamics, spectral element method, graphical processing units, message passing interface

Sammanfattning

Motiverat av slutet av Moores lag så har grafikprocessorer (GPU:er) börjat ersätta konventionella processorer som den huvudsakliga källan till beräkningingskraft i superdatorer. En utmaning i system med GPU-acceleratorer är kostnaden att överföra data mellan värdminnet och acceleratorminnet. På superdatorer är Message Passing Interface (MPI) en standard för kommunikation mellan beräkningsnoder. Nyligen stödjer många MPI-implementationer direkt användning av acceleratorminne som kommunikationsbuffertar, vilket kallas GPU-aware MPI.

En av de mest beräkningsintensiva applikationerna på superdatorer är nogranna datorsimuleringar av turbulenta flöden. Förbättrad prestanda i nogranna flödesberäkningar kan möjliggöra fall som idag är omöjliga, till exempel ett helt flygplan i luften.

I detta examensarbete jämför vi MPI-prestandan med värdminne och acceleratorminne, och demonstrerar hur GPU-aware MPI kan användas för att accelerera nogranna datorsimuleringar av inkompressibla flöden i spektralelementkoden Neko. På ett testsystem med NVIDIA A100 GPU:er finner vi att MPI-prestandan är liknande med värdminne och acceleratorminne. Detta gäller dock inte för meddelanden inom samma beräkningsnod i intervallet 1-64 KB vilka är betydligt långsammare med acceleratorminne, och över 1 MB vilka är betydligt snabbare med acceleratorminne. Vi finner också att prestandan av nogranna datorsimuleringar i Neko kan förbättras upp till 2,59 gånger genom användning av GPU-aware MPI i den så kallade gather–scatter-operationen, vilket undviker flera överföringar mellan värdminne och acceleratorminne.

Nyckelord

högprestandaberäkningar, beräkningsströmningsdynamik, spektralelementmetoden, grafikprocessorer, meddelandeförmedlingsgränssnitt

Acknowledgments

First I would like to thank my supervisors and mentors Dr. Niclas Jansson and Martin Karp for guiding me through the project and teaching me about computational fluid dynamics. I would also like to thank my examiner Prof. Stefano Markidis for introducing me to high-performance computing and helping me find this thesis topic. My colleagues at PDC Center for High Performance Computing have also been helpful, in particular, the assistance of Gert Svensson, Gilbert Netzer, Ragnar Sundblad, and Luca Manzari has been greatly appreciated. Mikael Öhman at C3SE has also been helpful in answering questions about the MPI library on Alvis. Finally, feedback from Sam Hedin and Henric Zazzi was valuable in finalizing the thesis manuscript.

The computations in this thesis were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE) and PDC Center for High Performance Computing at KTH Royal Institute of Technology, partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

Stockholm, June 2022
Jacob Wahlgren

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

CFD	Computational Fluid Dynamics
CG	Conjugate Gradient method
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GCD	Graphics Compute Die
GMRES	Generalized Minimal Residual method
GPU	Graphics Processing Unit
HIP	Heterogeneous Interface for Portability
HPC	High Performance Computing
HSMG	Hybrid Schwarz Multigrid
MPI	Message Passing Interface
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PGAS	Partitioned Global Address Space
RDMA	Remote Direct Memory Access
SEM	Spectral Element Method
SIMD	Single Instruction, Multiple Data

Chapter 1

Introduction

Supercomputers power large-scale simulations in science and engineering as well as training of large machine learning models. A modern supercomputer consists of many independent compute nodes that are connected by a high-speed network. Motivated by the end of Moore's law, specialized accelerators are replacing general-purpose processors as the main provider of computational power in emerging supercomputing architectures.

Today, graphics processing units (GPUs) are used as compute accelerators due to their high throughput. Seven of the top ten supercomputers use GPUs, among them, the world's fastest computer Frontier [1].

1.1 Motivation

In GPU-accelerated systems, a general-purpose central processing unit (CPU) controls program flow and submits computationally heavy tasks for the GPU accelerator to complete. In common terminology, the CPU is called the host while the GPU is called the device. The host and device have separate memories, with a cost associated with transferring data between them. To achieve high performance, data must be kept in device memory as much as possible to ensure it is readily available for computations on the GPU [2].

The architecture of a simple GPU-accelerated compute node is illustrated in Figure 1.1. The two GPUs provide the majority of the compute capacity, while the CPU handles initialization and program flow. The lines indicate connections between components. The CPU is connected to a network interface card (NIC) which is the link to the high-speed network for communication with other compute nodes in the cluster.

The standard programming interface for communication between compute

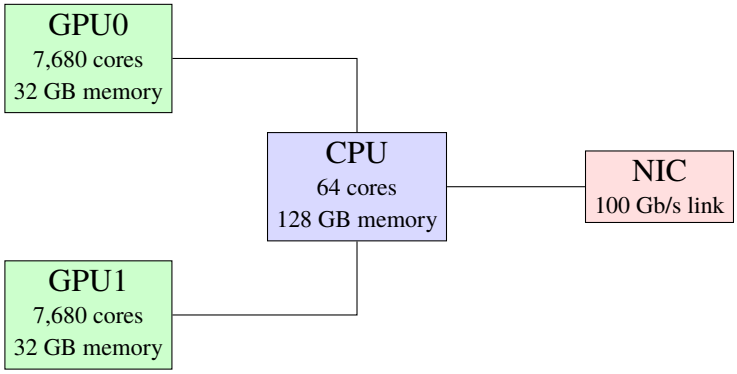


Figure 1.1: Illustration of a simple GPU-accelerated compute node architecture. Based on an AMD MI100 test node at PDC Center for High Performance Computing.

nodes on supercomputers is called Message Passing Interface (MPI). The MPI programming interface consists of functions such as `MPI_Send` to send a message to another process and `MPI_Recv` to receive a message from another process. Further details are provided in Section 2.1.

Traditionally MPI operates exclusively in host memory. However, with the rise of GPU accelerators many MPI libraries today support using device memory directly as communication buffers, which enables the use of GPU-optimized transport strategies. MPI libraries that support device memory buffers are known as GPU-aware. Examples of GPU-aware MPI implementations include OpenMPI and Cray-MPICH, which are used in this thesis.

Computational fluid dynamics (CFD) studies the flow of fluids such as air or water through computer simulation. In CFD, numerical methods are used to solve the Navier–Stokes equations that describe fluid flow. There are many applications of fluid simulations in science and engineering such as aerodynamics in the vehicle industry, blood flow in medicine, ocean currents in climate research, as well as fundamental research in fluid mechanics.

Accurate simulations of turbulent flows require high-fidelity methods and large computational resources. The level of turbulence is characterized by the Reynolds number Re. The computational complexity of a direct numerical simulation grows quickly with the Reynolds number: the number of grid points required for a direct numerical simulation is $O(Re^{9/4})$, and the number of operations is $O(Re^3)$ [3], [4]. Therefore direct numerical simulations of turbulence are performed on supercomputers. Improved performance in high-fidelity methods can enable more complex cases that are intractable today, such

as a complete simulation of an aircraft in flight.

The spectral element method is a numerical method that has successfully been used for large-scale high-fidelity fluid simulations. Tufo and Fischer were awarded the Gordon Bell Special Prize in 1999 for their work on the spectral element code Nek5000 [5], [6], which has been shown to scale to 1 million CPU cores [7]. The numerical methods used in Nek5000 are still highly relevant. However, the Fortran 77 code of Nek5000 is not suited for modern GPU-accelerated architectures [8]. Therefore new codes based on the spectral element method are being developed. In particular, this thesis is concerned with the Neko framework, which is currently developed by researchers at KTH Royal Institute of Technology [8].

1.2 Problem

Software for high-fidelity simulations of turbulence requires large computational resources. Therefore, new codes are being developed to leverage the power of modern supercomputers with GPU accelerators.

To achieve high performance in GPU-accelerated software, data transfers between host and device memory must be avoided [2]. A promising technology in this regard is GPU-aware MPI, which enables inter-process communication directly to and from GPU device memory. We aim to answer the following questions:

1. How does GPU-aware message passing with device memory buffers perform compared to conventional message passing from host memory?
2. How can GPU-aware message passing be leveraged to optimize large-scale fluid simulations in the spectral element method?

1.3 Purpose

The objectives of the project are:

- To optimize the scalability of the fluid solver Neko for NVIDIA and AMD GPU clusters.
- To investigate how GPU-aware message passing can be leveraged to accelerate scientific software.

The scientific results will be relevant both to users of fluid simulations and to researchers in high-performance computing. The Neko project is conducted in collaboration with the mechanics department at KTH, where high-fidelity fluid simulations are used for instance to study flow across aircraft wings [9]. For HPC research the findings can be used to help accelerate other scientific software.

The degree project is related to sustainability in two ways. First, by accelerating scientific computations, the amount of energy used for computational science may be reduced. Nine of the ten most energy-efficient supercomputers in the world feature GPUs [10]. Second, more efficient fluid solvers enable simulations at higher Reynolds numbers and larger domains than currently possible.

1.4 Goals

The goals of the degree project are:

- To conduct a study of scientific literature and technical manuals to learn how GPUs, and especially GPU-aware MPI, can be efficiently leveraged in scientific software.
- To develop an optimized GPU-aware communication backend for Neko, and make it available in a future release of the software. To achieve this goal, familiarity with the existing codebase, and proficiency in the programming languages Fortran and C++, as well as the MPI, CUDA, and HIP programming interfaces is required.
- To characterize the performance of the optimized GPU implementation in comparison to the baseline implementation. To achieve this goal, computing resources on a GPU cluster as well as a relevant test case is required.

1.5 Research methodology

A qualitative study of related works is conducted to collect experiences for the design of a new GPU-aware communication backend. Quantitative timing experiments are conducted to characterize the performance of GPU-aware communication as well as the new communication backend. Isolated benchmarks of the communication backend, as well as complete simulation

runs, are used. The benchmarks are run across various numbers of nodes to characterize the parallel scaling behavior.

1.6 Delimitations

Since the project is based on the existing Neko software, there is no need to develop a complete fluid solver from scratch. The codebase already includes a baseline GPU implementation, using conventional message passing from host memory.

The study is focused on the gather–scatter operation, which is the main communication kernel in the method. The gather–scatter operation is described in detail in Section 2.3. Other communication operations, such as global reductions and file output are left as future work.

The performance is only evaluated on a single cluster with NVIDIA A100 GPUs due to limited hardware availability. On other systems, the performance characteristics may differ.

Neko supports simulations in both single-precision and double-precision floating-point numbers. This project is limited to the double-precision version since it is the most used in practice.

1.7 Structure of the thesis

The thesis is organized as follows. The relevant background information is presented in Chapter 2, the research methods are described in Chapter 3, the implementation of the accelerated communication is described in Chapter 4, the results of the experiments are presented and analyzed in Chapter 5, and the conclusions and future work are presented in Chapter 6.

Chapter 2

Background

In this chapter, a survey of the relevant background information and context of the thesis topic is presented. The field of high-performance computing including message passing and GPU acceleration is introduced in Section 2.1. Fluid mechanics, particularly computational fluid dynamics, is introduced in Section 2.2. The numerical methods used in Neko are described in the section on the spectral element method, Section 2.3. Finally, related works are presented in Section 2.4, and a summary is provided in Section 2.5.

2.1 High-performance computing

In high-performance computing (HPC), highly parallel computer systems are used to solve computationally demanding problems [11]. The most advanced computer systems are called supercomputers and are composed of many compute nodes connected by a high-speed network. A single application can run on many nodes that use message passing to exchange data with each other.

Parallelization is commonly a trade-off between total execution time and resource usage efficiency [11]. When more compute nodes are used, the ratio of time spent in communication to time spent in computation increases, and thus the efficiency decreases. However, as more compute nodes are used, the total execution time also decreases.

The parallelization trade-off is illustrated by Amdahl's law [12], which is a model of the speedup S achieved by using P parallel processors. If a program has a perfectly parallelizable fraction p and a serial fraction $1 - p$ then the speedup according to Amdahl's law is

$$S = \frac{1}{(1-p) + p/P}. \tag{2.1}$$

There is a diminishing return in S of increasing P. Asymptotically the speedup is limited by the serial fraction.

$$\lim_{P \to \infty} S = \frac{1}{(1-p)} \tag{2.2}$$

The efficiency of a computation can be measured using the parallel efficiency η. Parallel efficiency relates the execution time T_P using P parallel processors to the execution time using just a single processor T_1. Depending on the platform, P can denote the number of GPUs, CPU cores, etc.

$$\eta = \frac{T_1}{PT_P} \tag{2.3}$$

End-users of high-performance software must decide which efficiency is acceptable in their case. By improving the parallel efficiency, the total execution time for end-users may be improved, as well as enabling larger problem sizes.

2.1.1 Message passing

The standard interface for communication between compute nodes in high-performance computing is called Message Passing Interface (MPI). An application using MPI is divided into many distributed processes known as ranks. Each process can send messages to the other processes by using a set of functions supplied by the MPI library. In a cluster without accelerators, each rank may be assigned to a CPU core. In a cluster with GPUs, each rank may be assigned to a GPU instead.

Depending on the location of the sending and receiving ranks, the MPI library may select different underlying transport methods. For instance, if the ranks are assigned to two CPU cores on the same node, the MPI library can simply use a `memcpy` from the sender's send buffer to the receiver's receive buffer. If the ranks are assigned to two CPU cores on different nodes, the MPI library must send the data in a packet on the high-speed network.

When GPU accelerators were first introduced to high-performance computing, programs using MPI had to copy data back and forth between host memory and device memory, since the computations were done in device memory but the communication device memory is expensive.

The term GPU-aware is used to refer to MPI implementations that support communication buffers in GPU device memory. With GPU-aware MPI, there is no longer a need to copy data between host and device

Listing 2.1: Code example with MPI from host memory.

```
for 1..N {
    d_y = gpu_compute(d_x)
    h_y = device_to_host(d_y)
    h_z = mpi_communicate(h_y)
    d_x = host_to_device(h_z)
}
```

Listing 2.2: Code example with GPU-aware MPI from device memory.

```
for 1..N {
    d_y = gpu_compute(d_x)
    d_z = mpi_communicate(d_y)
    d_x = d_z
}
```

memory for communication. It allows the MPI implementation to optimize communication by using GPU-specific transport strategies, for instance, inter-process communication (IPC) libraries from GPU vendors or remote direct memory access (RDMA) to the NIC.

An example of how GPU-aware can be utilized is shown in Listings 2.1 and 2.2. In this code, a leading `d_` indicates that the variable is stored in device memory, and a leading `h_` indicates that the variable is stored in host memory. Both codes implement the same algorithm, with an outer loop that alternates between computations on the GPU, and communication using MPI. In Listing 2.1, data has to be moved between host and device memory, while Listing 2.2 leverages GPU-aware MPI to do the communication in device memory. Higher performance can be expected from Listing 2.2 since data movement is avoided.

The concept of GPU-aware MPI was first introduced in the MVAPICH2 project in 2011 [13]. This implementation used pipelined transfers between host and device to hide network latency. In 2013, NVIDIA's GPUDirect was used in MVAPICH2 to accelerate the GPU-aware MPI implementation by avoiding transfers between host and device completely [14]. The authors found that their design could improve the latency by up to 69% and the bandwidth by up to 2x for small messages. Today GPU-aware features are included in many MPI implementations, such as OpenMPI and Cray-MPICH.

Emerging high-performance node architectures like the AMD CDNA 2 Flagship HPC Topology with AMD MI250X GPUs are optimized for direct

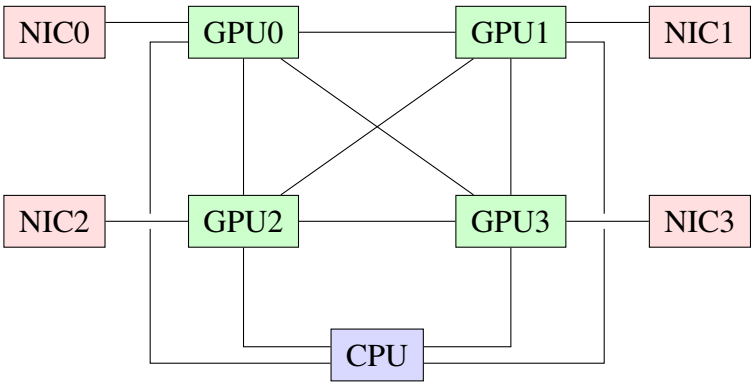


Figure 2.1: Node architecture of the AMD CDNA 2 Flagship HPC Topology with AMD MI250X GPUs [15]. Note that the CPU is not directly connected to any NIC.

transfers from device memory to NIC [15]. It is featured in the world's first exascale supercomputer Frontier [16], and also the Dardel supercomputer at KTH Royal Institute of Technology in Sweden [17]. The node architecture is illustrated in Figure 2.1.

2.1.2 GPU acceleration

Starting with NVIDIA's introduction of CUDA in 2007 [18], GPUs (graphics processing units) have evolved from specialized graphics processors to generally programmable computing accelerators. Modern GPUs are optimized for massive throughput and are especially useful in the scientific computing and machine learning domains [15].

A GPU-accelerated application has two parts: a program running on the host processor (CPU), and programs known as kernels that run on the device (GPU). The host sends commands to the device to transfer data between host and device memory, and to launch kernels on the device. High throughput is achieved by overlapping compute operations with memory access operations [19].

A GPU is a massively parallel processor consisting of many compute units. Each compute unit contains a register file and an execution pipeline with scalar and vector instructions, as well as matrix instructions in recent models [15]. An AMD Instinct compute unit has four 16-wide Single Instruction Multiple Data (SIMD) units that together can process 64 double-precision floating-point numbers per cycle, in what is called a wavefront [20]. An AMD MI100 contains 120 compute units executing in parallel [20]. Depending on the

number of registers occupied by each wavefront, an MI100 compute unit can handle up to ten wavefronts at a time [20].

A GPU has a hierarchical memory system. At the top is a large but slow global memory that can be accessed by all compute units, as well as a smaller global L2 cache. Further on, each compute unit has a fast L1 cache as well as a programmable local data store (also known as shared memory) for data shared between the lanes within a wavefront [15]. If a kernel has a high operational intensity it may be beneficial to load data in global memory to the local data share at the start of the kernel [19].

Software written for regular CPU processors cannot be compiled to run on a GPU. Code running on GPUs is written using a special programming interface in C++, for NVIDIA GPUs the CUDA interface is used and for AMD GPUs the HIP interface is used.

2.2 Fluid mechanics

Fluid mechanics is devoted to the study of fluid motion in physics. Fluid flow is described by the Navier–Stokes equations. Due to the non-linearity of the equations, fluid mechanics often have to be studied using computational methods [21]. The study of the Navier–Stokes equations using computational methods is known as computational fluid dynamics (CFD).

The Reynolds number Re is a non-dimensional quantity used to characterize a flow. The Reynolds number indicates the relative influence of inertial forces to viscous forces. The higher the Reynolds number, the more turbulent and unstable a flow is. At high Reynolds numbers, the physics becomes very complicated and chaotic, with turbulent structures at spatial scales spanning several orders of magnitude [22]. As famously put by fluid dynamics pioneer Richardson in 1922: “Big whirls have little whirls that feed on their velocity, and little whirls have lesser whirls and so on to viscosity” [23].

In general, a fluid simulation problem is defined by a domain Ω, boundary conditions on $\partial\Omega$, and an initial condition at time $t = 0$. The problem is to determine how the flow develops using the Naiver–Stokes equations.

2.2.1 Applications of computational fluid dynamics

This section describes some examples of important applications of computational fluid dynamics [21]. Historically, the most prominent applications have been in aerospace engineering. Automotive applications are also increasingly

moving to computational methods in addition to wind tunnel experiments. In modern biomedicine, fluid simulations are used in vascular dynamics to simulate blood flow for diagnosis and surgical planning. Simulations also play an important role in the energy sector, both in ensuring the safety of nuclear reactors and the design of efficient wind farms. Other applications include oceanography, metrology, and heat-exchanger design [22].

2.2.2 The Navier–Stokes equations

The Navier–Stokes equations are a set of nonlinear partial differential equations describing the flow of viscous fluids [22]. The equations have been known since the 19th century, but they are still not fully understood by mathematicians. The existence of smooth (physically reasonable) solutions to the incompressible Navier–Stokes equations in three dimensions is one of the Clay Institute's $1 million Millennium Prize problems [24].

Let u be the velocity, p the pressure, f a given volume force, and $Re = UL/\nu$ the Reynolds number, where U is a reference velocity, L is a reference length, and ν is the kinematic viscosity. The following equations show the non-dimensional form of the Navier–Stokes equations [8].

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\nabla p + \frac{1}{Re}\nabla^2 u + f \tag{2.4}$$

In many technically important cases, the flow can be regarded as incompressible, meaning that the density of each material particle is constant [25, p. 38]. An incompressible flow is modeled by the incompressible Navier–Stokes equations which additionally include the divergence-free condition below.

$$\nabla \cdot u = 0 \tag{2.5}$$

2.3 Spectral element method

Many mathematical problems are too complex to solve analytically, therefore numerical methods are used to find approximate solutions. As previously noted, one such problem is the solution of the incompressible Navier–Stokes equations (Equations 2.4 and 2.5). This section describes the numerical methods used for incompressible fluid simulations in Neko.

In general, the method to solve a system of partial differential equations, such as the Navier–Stokes equations, is to first reformulate the continuous problem as a discrete problem. The domain is modeled using a finite grid and

time is modeled as a finite set of time steps. The discretization enables posing the problem as a set of linear equations, resulting in one linear system (matrix equation) per time step and unknown. In the three-dimensional Navier–Stokes equations there are four unknowns: pressure and velocity in the three spatial dimensions. Thus in each time step, four linear systems are solved.

In the spectral element method, the grid is defined by a mesh of elements. In Neko, the elements are hexahedra (e.g. cubes). On each element, a polynomial basis of order N is used. There are thus $N+1$ nodal points in each direction on each element. An example of a spectral element mesh is shown in Figure 2.2. All computations are performed in the unassembled form. By using the gather–scatter operation the assembled form is avoided entirely.

The gather–scatter operation $w = Q^T Q u$ is key to the thesis topic. It is described in detail in the following subsection, but a simple explanation is provided here. The vector u has one or more components per point in the domain. To ensure that the solution is continuous, we must ensure that all components that represent the same point have the same value. This is done using the gather–scatter operation, which sums values representing the same points together. For instance, if u_i and u_j represent the same point, then $w_i = w_j = u_i + u_j$. In practice, the vectors u and w are distributed over all the MPI ranks, therefore communication between the ranks is required to implement the operation.

2.3.1 Discretization method

For discretizing the problem, a spectral element method pioneered by Patera [26] is used. The method is known as the $\mathbb{P}_N - \mathbb{P}_N$ method in Nek5000 [27], and is described in [8] and in more detail in [28]. The monograph [22] is a complete review of spectral element methods.

The Navier–Stokes equations are separated into four independent implicit problems per time step [28]. The pressure in the next time step is given by a Poisson equation

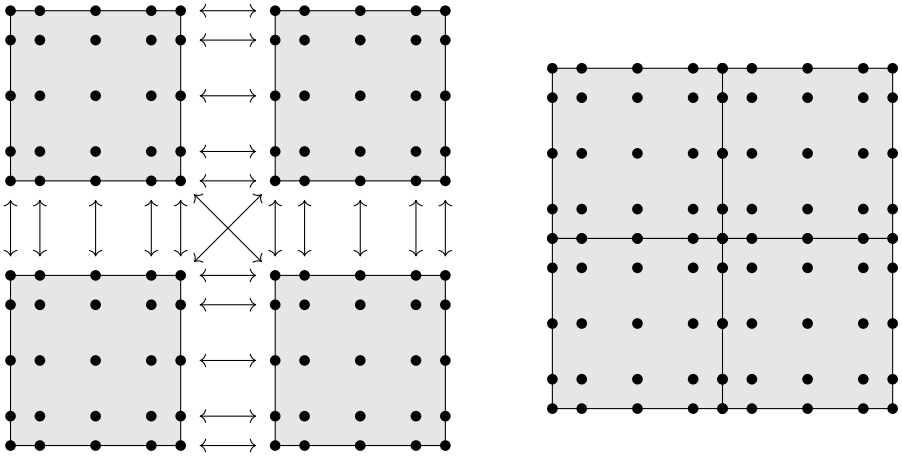
$$-\nabla^2 u = f$$

while each component of the velocity is given by an inhomogeneous Helmholtz equation

$$-\nabla^2 u + qu = f$$

with constants q, f. The unknown u represents the pressure or a velocity component in our domain $\Omega \subset \mathbb{R}^3$, i.e. $u : \Omega \to \mathbb{R}$.

Decoupling the pressure and velocity components allows us to solve the



(a) Unassembled form. (b) Assembled form.

Figure 2.2: Illustration of a two-dimensional mesh with four square elements and polynomial order $N = 4$. The assembled form represents the complete domain. In the unassembled form, the elements have been disconnected so they can be processed in parallel. Arrows indicate shared points that are summed using the gather–scatter operation to ensure continuity.

problems using different techniques best suited for each problem (explained in the next section on iterative methods). Also note that the Poisson equation is a special case of the Helmholtz equation ($q = 0$), which means that the implementation of the discretization can be shared in practice [7].

First, we cast the partial differential equations (a Poisson or Helmholtz equation plus boundary conditions) into their weak form, namely to find a function u in the solution space $V \subset \Omega \to \mathbb{R}$ such that

$$\mathcal{A}(u, v) = \mathcal{L}(v) \quad \forall v \in V$$

where $\mathcal{A}$ is a bilinear form, and $\mathcal{L}$ is a linear form. The solution space V is limited to continuous functions.

The domain Ω is partitioned into E hexahedral elements. On each, element we use orthogonal Legendre polynomial basis functions L_{ijk} of degree N based on Gauss–Lobatto–Legendre quadrature points. Each local basis function φ^e_{ijk} corresponds to some piecewise global basis function φ_l:

$$\varphi_l(x) = \begin{cases} \varphi^e_{ijk}(x) & \text{if } x \in \Omega_e \\ 0 & \text{otherwise} \end{cases}$$

for some global index $1 \leq l \leq n$, element $1 \leq e \leq E$, and local coordinates $1 \leq i, j, k \leq N + 1$. This results in a piecewise polynomial approximation space $V_n \approx V$ with $n = E(N + 1)^3$ basis functions.

Let $x_l = x^e_{ijk}$ denote the corresponding nodal points. Note however that the global basis is not completely orthogonal. At the edge between two elements, the local basis functions share the same nodal point, and thus the number of unique nodal points m is less than the number of basis functions n.

With the approximation space defined, we now discretize the weak form by Galerkin projection onto V_n. We obtain the discrete unknown $\underline{u} = (u_1, u_2, ..., u_n)^T$, the discrete bilinear form A (the Helmholtz operator), and the load vector b [22, p. 175]. Note that $A = K + qM$ where K is the stiffness matrix and M is the (diagonal) mass matrix.

$$\begin{aligned} u &= \sum_{i=1}^{n} u_i \varphi_i \\ A_{ij} &= \mathcal{A}(\varphi_j, \varphi_i) \\ b_i &= \mathcal{L}(\varphi_i) \end{aligned}$$

The discrete problem is to find the coefficients $\underline{u}$ (also known as degrees of freedom, dofs) to satisfy the following matrix equation.

$$A\underline{u} = b$$

While the n-dimensional vector b is manageable, the n^2 components of the matrix A pose a challenge. It is infeasible to assemble and store A explicitly. Luckily A is sparse, since $A_{ij} = 0$ if φ_i and φ_j belong to different elements and $x_i \neq x_j$. In fact, A is composed of the entries for the shared nodes, plus dense $(N+1)^3 \times (N+1)^3$ submatrices A^e where the basis functions are from the same element.

Additionally, we need to restrict the problem by enforcing continuity of u across element boundaries by equating the coefficients at shared points. In practice, we work with the unassembled matrix $A_L = \mathrm{diag}(A^1, A^2, ..., A^E)$ plus the gather–scatter operator $Q^T Q$. The gather matrix Q maps from n local coefficients $\underline{u}_L$ to M globally unique coefficients by summing the contributions at shared nodes. The scatter matrix Q^T copies the M unique coefficients back into n coefficients. Let $\underline{u}_L$ be some coefficient vector. Then $\underline{u} = Q^T Q \underline{u}_L$ represents a continuous version of $\underline{u}_L$ where the values at shared nodes have been averaged. This operation is also called direct stiffness summation. We modify the matrix equation to include A_L and the continuity

constraint.

$$Q^T Q A_L \underline{u}_L = b$$

Using this formulation we can partition the elements onto a set of parallel processors such that computations involving A_L are local to the processor, and only the gather-scatter operator $Q^T Q$ requires communication between the processors. In practice, the matrices Q and Q^T need not be assembled, the operator can be implemented efficiently by maintaining a two-way mapping between local degrees of freedom and global unique nodal points.

The size of the dense local element matrix A^e is $(N+1)^6$, which quickly becomes infeasible for higher-order (spectral) methods. It is common to use polynomial degrees in the range 7–11. Taking advantage of the local tensor-product Lagrange basis, we can factorize A^e using $(N+1)^3$ geometric factors per element, and $3(N+1)^2$ Lagrange derivatives [22]. Thus our factorization of A reduces the problem size from $O(n^2) = O(E^2 N^6)$ to $O(EN^3)$.

2.3.2 Iterative methods

To solve the sparse linear systems $Ax = b$ described in the previous section, iterative methods are employed. In an iterative method, successive approximations of x are generated in a loop until the required accuracy has been reached [29]. The rate of convergence depends on the condition number, i.e. how sensitive x is to small perturbations in A or b. For symmetric systems, the condition number can be defined as the ratio of the largest to the smallest eigenvalue of A, and this is a good approximation for nonsymmetric systems as well [22]. The convergence rate can be improved by using a preconditioner M and solving $M^{-1}Ax = M^{-1}b$ instead.

In Neko, the Krylov subspace methods conjugate gradient (CG) and generalized minimal residual (GMRES) are used [8]. Each iteration of these methods requires one or more calls to the gather–scatter operation.

The linear systems differ in condition number [8]. The velocity systems are well-conditioned and thus a simple Jacobi preconditioner is used. The Jacobi preconditioner is computed by assembling the diagonal $M = \mathrm{diag}(A)$. The pressure system is ill-conditioned and requires an advanced preconditioner like a Hybrid-Schwarz Multigrid method (HSMG) [30], [31]. In the HSMG preconditioner, the system is solved on a coarser grid with linear elements ($N = 1$) and then interpolated to the full grid (e.g. $N = 7$) [8].

The number of iterations required to solve the systems also depends on the quality of the initial guess $x^{(0)}$. The fact that the operator A is constant over all the time steps can be leveraged to improve $x^{(0)}$. By storing a number of old

problems b and solutions x, a space is constructed, and the new b is projected onto it to obtain the new initial guess [32].

2.4 Related works

The Neko code is presented and analyzed in [8]. The GPU backend already takes advantage of some optimizations introduced for the NEC SX Vector Engine backend [33].

There are several other efforts to accelerate spectral element fluid simulations. NekRS is another modern SEM code that supports GPU acceleration using the OCCA abstraction layer [28]. Part of Nek5000 has been accelerated using OpenACC directives [34].

The NekRS code can take advantage of GPU-aware MPI in their gather–scatter implementation [28]. At initialization time different gather–scatter implementations are tested and the fastest is chosen. This has been further studied in the proxy application hipBone, where efforts are also made to overlap gather–scatter with computation [35].

High-performance GPU-aware communication has previously been studied by Dryden, Maruyama, Moon, *et al.* in the context of deep neural networks [36]. They find that global reduction communication (allreduce) is a significant bottleneck for training at scale, and propose a communication library called Aluminum to provide asynchronous communication backed by CUDA, NCCL, and MPI. In the bandwidth-limited regime, the NCCL (i.e. GPU-aware) backend performs best, while in the latency-limited regime, the NCCL backend performs better up to 32 GPUs and the host MPI backend performs better at 64 GPUs and above. This is attributed to the fact that NCCL is optimized for bandwidth but also can take advantage of direct GPU transfers and knowledge of the node-local topology. Unfortunately, Aluminum does not fully support GPU-aware MPI or AMD hardware. The library provides a model for overlapping communication with computation similar to CUDA streams, which is not as useful in the context of numerical codes where the computations are usually dependent on data from the preceding communication.

The developers of PETSc, a general-purpose library for scalable numerical solvers, are also faced with the challenges posed by upcoming exascale architectures [37]. The library is migrating from direct MPI calls to a generalized communication module called PetscSF based on a star forest model [38]. A star forest is a graph with multiple depth 1 trees. This model is similar to the gather–scatter model in Neko. Internally the PetscSF

module takes advantage of GPU-aware MPI when available, in addition to other methods like NVSHMEM.

2.5 Summary

In high-performance computing, highly parallel computer systems are studied. These systems are composed of many compute nodes that communicate, commonly by message passing using the MPI interface. Many modern supercomputers feature GPU accelerators, which provide large computational capacity.

In computational fluid dynamics, numerical methods are used to understand the behavior of fluid flow. Incompressible flow is described by a set of partial differential equations called the incompressible Navier–Stokes equations.

The spectral element method is a numerical method used to solve the incompressible Navier–Stokes equations on parallel computers. A key operation in the spectral element method is the so-called gather–scatter operation which is the main communication kernel.

Chapter 3

Methods

In this chapter, the research methods used in the degree project are described. An overview of the research process used is presented in Section 3.1, the verification method is described in Section 3.2, the method of application performance evaluation is described in Section 3.3, the system performance evaluation method is described in Section 3.4, the gather–scatter benchmark method is described in Section 3.5, and the experimental setup is presented in Section 3.6.

3.1 Research process

The first step of the research process was to learn about computational fluid mechanics, numerical methods, and high-performance computing. The focus was particularly on the spectral element method, GPU programming, and MPI communication. The next step was to establish a verification and performance baseline using the current Neko version. Then profiling, development, and testing of new code were conducted. When the code was finished, final verification and performance experiments were run. The results were then analyzed and compared to the baseline. To aid the analysis, additional system performance tests were then conducted.

3.2 Verification method

To verify the accuracy of the code, a previously well-studied simulation case is used. The problem is a direct numerical simulation of the Taylor–Green vortex at $Re = 1600$ [39, Problem C3.5]. This problem is designed to test the accuracy and performance of high-order methods on the direct numerical

simulation of the transition from laminar to turbulent flow. The initial condition is simple, while the final state is highly chaotic. This benchmark case has previously been used to validate the accuracy of the CPU and Vector Engine backends of Neko [8]. The simulation runs from time $t = 0$ to $t = 20$.

We capture the enstrophy $\mathcal{E}$ at regular intervals, which is a scalar integral quantity related to the vorticity ω. The vorticity field describes the tendency for fluid to rotate. Since the enstrophy is very sensitive to solver accuracy it is suitable for verification. For incompressible flow, it can be computed as

$$\mathcal{E} = \int_{\Omega} |\omega|^2 \, dx. \tag{3.1}$$

The same simulation is run for both the baseline and the optimized version. The results are then compared to reference data from direct numerical simulations [39]. Due to limited computing resources, the case is run at a lower resolution of 32^3 elements and 224^3 grid points, resulting in an implicit large eddy filter. The reference data resolve all scales using 512^3 grid points. While this means that the verification case is not equivalent to the reference case, it still constitutes an approximate solution.

The time step is length 10^{-3} and the order is 7. We use the coarse grid pressure preconditioner and dealiasing. The verification is performed using a varied number of GPUs to ensure correctness across different network topologies.

3.3 Application performance evaluation

Two simulation cases are used to evaluate the impact of an accelerated gather–scatter operation on overall application performance. For both cases, a strong scaling study is conducted and the time per time step and parallel efficiency is measured. The time per time step is read from the output log file and the average and standard deviation are computed using an AWK script.

The smaller case tests the scaling from 1 to 12 GPUs. The simulation case used for verification is used in this case as well. The accelerated gather–scatter operation is used for $N > 1$. For this case, we measure the performance indicators in the most computationally intensive region $8 \geq t \geq 10$, since in practice the execution time of real cases is dominated by such computationally intensive regions. The computational cost can be measured by the number of pressure solver iterations required per time step, which for this case is shown in Figure 3.1.

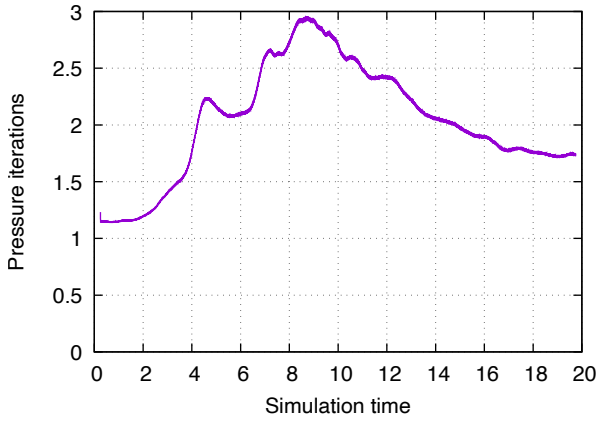


Figure 3.1: The number of iterations of the computationally expensive pressure solver in the verification case. A low pass filter has been applied to show a rolling average.

The larger case tests the scaling from 16 to 128 GPUs. For this case, 16 GPUs is the smallest possible number due to memory limitations, and 128 GPUs is the largest number of GPUs in a single run on the test system. The case is a larger version of the Taylor–Green vortex at $Re = 5000$. This has been used for strong scaling tests of Neko previously [8]. We use the same parameters as [8] with time step $5 \cdot 10^{-4}$, order 9, 64^3 elements, and no dealiasing. Due to limited computing resources, the only first 1000 time steps are computed. Thus the simulation does not have time to reach the most computationally costly region.

To characterize the parallel performance we compute the parallel efficiency (Equation 2.3). For the larger case, the following formula is used since the memory requirement prevents running on one GPU.

$$\eta = \frac{8T_8}{PT_P} \tag{3.2}$$

3.4 System performance evaluation

To compare the performance of GPU-aware MPI with device memory to MPI from host memory, a ping-pong benchmark is used. In this benchmark, the MPI performance is tested in isolation. The benchmark is called `osu_latency` and is part of the OSU Micro-Benchmarks suite, version 5.9 [40]. The test measures the average time to send messages up to 4 MB in size. We run it both between two GPUs on the same node and two GPUs on

different nodes. In both cases, the test results are averaged over ten different node configurations. For the intra-node test, GPU0 and GPU1 were used. For the intra-node test, GPU0 on each node was used. Performance may differ for other GPUs on the node depending on the interconnect topology.

Additionally, the time to copy data between host and device memory is measured. For powers of two up to 4 MB, a data buffer is copied from host memory to device memory and back using `cudaMemcpy`. Below 512 KB 10,000 warmup iterations and 10,000 timed iterations are used, and above 1,000 iterations are used. Average time per copy call is reported. Time is measured at the start and end using the Linux system call `clock_gettime(CLOCK_MONOTONIC)`.

3.5 Gather–scatter operation benchmark

For a deeper understanding of the performance characteristics, an isolated benchmark of the gather–scatter operation is used. The benchmark consists of 1000 calls to the gather–scatter operation (`gs_op`) on a vector. The polynomial order is varied to reflect different parts of the application. The orders 7 and 11 represent the main solver, while the orders 1 and 3 represent the coarse grid solver of the pressure preconditioner. At higher orders, more data is processed and the operation takes longer. At lower orders, there is less data and the operation is faster. However, the lowest order operation is called roughly ten times as often as the higher-order operations in the full application due to the construction of the coarse grid preconditioner.

A mesh with 64^3 elements is used and has been pre-partitioned to achieve a realistic balance of internal to external points at different numbers of GPUs and polynomial orders. The benchmark is run for 8 to 128 GPUs and the time per operation is measured using the function `MPI_Wtime`. The average and standard deviation are reported.

3.6 Experimental setup

Verification and performance measurement experiments are carried out on the Alvis supercomputer at C3SE at Chalmers Technical University. The used compute nodes have four NVIDIA A100 40GB GPUs, and two Intel Xeon Gold 6338 CPUs with 256 GB DDR4 memory. The GPUs are connected by a bonded NVLink interface for intra-node communication, and the node has two Mellanox ConnectX-6 200 Gb/s network interface cards (NICs). A

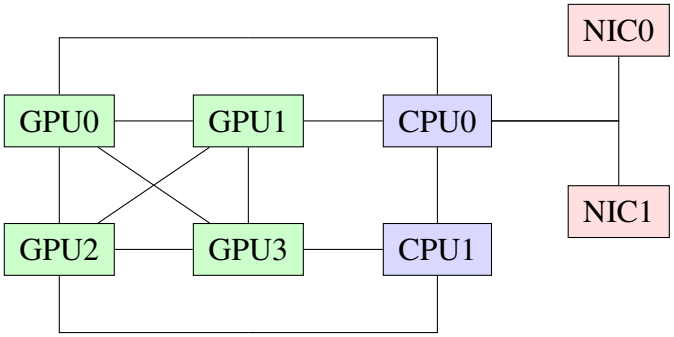


Figure 3.2: Alvis A100 compute node diagram. Connectivity captured with the command `nvidia-smi topo -m`.

Listing 3.1: Sample job script for Alvis to run Taylor–Green vortex case on 2 nodes with 4 GPUs each.

```
1  #!/bin/bash -e
2
3  #SBATCH -A XXX
4  #SBATCH --gpus-per-node=A100:4
5  #SBATCH -N 2
6  #SBATCH -t 1:00:00
7  #SBATCH --ntasks-per-node=4
8  #SBATCH --gres=gpuexcl:1
9
10 module load fosscuda
11
12 mpirun ./ompi_launch ./neko tgv.case
```

diagram of the compute node is shown in Figure 3.2. The code is compiled using GCC 10.2 and CUDA 11.1.1, compute capability 8.0, with NVIDIA driver 510.39.01. The MPI library is OpenMPI 4.0.5 with UCX 1.9.0.

To execute a program on the cluster, the batch job system SLURM is used together with the `mpirun` command. A sample job script is shown in Listing 3.1. To assign GPUs to MPI ranks the script in Listing 3.2 is used. We rely on SLURM and OpenMPI to pair GPU devices to CPU cores. Better performance may be achievable by making the pairing explicitly with consideration of NUMA domains.

Additionally, verification of the AMD/HIP version is performed on a test system at PDC Center of High Performance Computing at KTH Royal Institute of Technology. This system has two AMD MI100 GPUs with 32 GB memory each, one 64-core AMD EPYC 7742 CPU, and 128 GB RAM. The code was

Listing 3.2: Script for assigning GPUs to MPI ranks (`ompi_launch`).

```
1 #!/bin/bash
2
3 export CUDA_VISIBLE_DEVICES=${
     OMPI_COMM_WORLD_LOCAL_RANK}
4 exec $*
```

	Alvis compute node	AMD test node
GPU model	4x NVIDIA A100 40GB	2x AMD MI100 32GB
CPU	2x Intel Xeon Gold 6338	1x AMD EPYC 7742
Memory	256 GB	128 GB
GPU interconnect	Bonded NVLink	–
NIC	2x Mellanox ConnectX-6	–
Fortran compiler	GCC 10.2	Cray 13.0.0
GPU runtime	CUDA 11.1.1	ROCm 4.5.2
MPI	OpenMPI 4.0.5	Cray-MPICH 8.1.12

Table 3.1: Setup of the two test systems.

compiled with Cray Fortran 13.0.0 and ROCm 4.5.2. The MPI library is Cray-MPICH 8.1.12. The node architecture is illustrated in Figure 1.1. The setups of both test systems are also listed in Table 3.1.

The baseline Neko implementation is version 0.3.1 with the CUDA device backend for the NVIDIA system and the HIP device backend for the AMD system. The accelerated version is enabled by setting the configuration flag `--enable-device-mpi`. The Neko code base is hosted on Github, and can be found online at `https://github.com/ExtremeFLOW/neko`.

Chapter 4

Accelerated gather–scatter communication

In this chapter, the implementation of the accelerated gather–scatter operation leveraging GPU-aware MPI is described. Alternative strategies are discussed in Section 4.1 and the implementation details are described in Section 4.2.

4.1 Communication strategy

To implement the gather–scatter operation, we must decide on a strategy for communication between ranks. The communication is used to exchange the values at element boundaries. For instance, assume rank A owns element E_1 and rank B owns element E_2. If elements E_1 and E_2 are touching, then rank A must send the values of E_1 along their intersection to rank B, and vice versa.

We use a strategy similar to the baseline implementation, where data is packed into contiguous buffers and transferred using non-blocking send and receive operations with MPI. However, instead of copying the data from device memory to host memory, and packing it using the CPU, we do all operations in device memory using GPU-aware MPI. This allows portability across many types of systems since MPI is a well-established standard. The MPI library is free to choose the transport (i.e. CUDA IPC, GPU RDMA, etc.) depending on message size and destination.

An alternative implementation could use MPI datatypes of type indexed block to avoid explicit pack and unpack kernels. This could simplify the code and potentially provide better performance if well supported by the MPI implementation. MVAPICH2 has optimized support for non-contiguous datatypes in GPU memory [41], but in OpenMPI non-contiguous datatypes

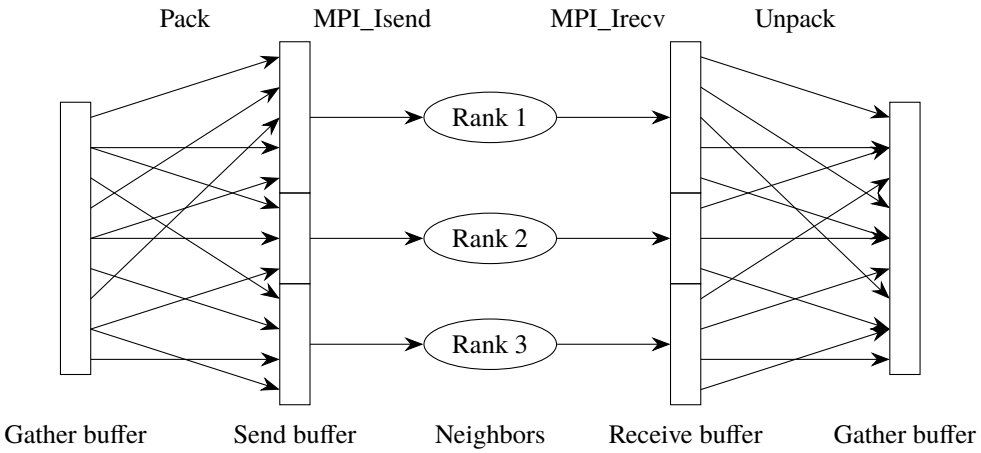


Figure 4.1: Illustration of the data flow in the accelerated gather–scatter implementation. All ranks execute the operation in parallel.

incur a high overhead [42].

Other communication libraries include NVIDIA's NCCL and AMD's RCCL which implement an interface similar to MPI but adapted specifically for GPU communication. However, RCCL has only "initial support" for point-to-point communication [43]. In the completely different Partitioned Global Address Space model, there is NVSHMEM from NVIDIA [44], and an "experimental prototype" ROC_SHMEM from AMD [45].

4.2 Implementation

Two device memory areas are allocated for send and receive buffers. A vector stores the mapping from the gather vector to the send buffer, and another vector stores the mapping from the receive buffer back to the gather vector. Two kernels are implemented in both CUDA and HIP, to pack and unpack the communication buffers. The data flow is shown in Figure 4.1.

A high-level simplified version of the implemented code is shown in Listing 4.1. The comments denote operations that were already implemented in Neko. First, the non-blocking receive operations are started. Then the shared points are gathered locally and then packed into the send buffer based on which neighbors share which points. Then `device_sync` is used to wait for the pack kernel to finish. This function blocks the host until all currently running GPU kernels are completed. Then the non-blocking send operations are initiated. During the non-blocking communication, the internal points are

gather–scattered. We wait for the receive operations to finish and then invoke the unpack kernel to sum the received data into the gather buffer. We wait for the send operations to finish. A final synchronization is done to avoid a race condition.

Simplified versions of the pack and unpack kernels in CUDA/HIP are shown in Listings 4.2 and 4.3 respectively. Note that indices from the mapping have to be offset to translate between Fortran semantics of 1-based indexing and C semantics of 0-based indexing. The pack kernel is a simple indirect copy operation. During unpacking the received data is summed into the gather buffer. If a point is shared with multiple neighbors, it is marked with a negative index, and summed using an atomic operation. Using atomics for all points would be expensive at higher polynomial orders since atomics are comparably expensive and duplicated points are uncommon.

While the main code is written in Fortran, the calls to MPI using device buffers are made from wrapper functions written in C. Attempting to pass a `type(c_ptr)` pointing to device memory to the Fortran MPI interface leads to crashes on the AMD test system.

Listing 4.1: Simplified code for gather–scatter using GPU-aware MPI. The `buf` and `map` arrays are stored in device memory. The `kernel` functions are executed on the GPU.

```
1  do i = 1, num_neighbors
2      call MPI_Irecv(recv_buf, offset(i),
3                     num_points(i), source(i))
4  end do
5
6  ! ... Gather shared points to gather_buf ...
7
8  call pack_kernel(gather_buf, send_buf, send_map)
9  call device_sync()
10
11 do i = 1, num_neighbors
12     call MPI_Isend(send_buf, offset(i),
13                    num_points(i), dest(i))
14 end do
15
16 ! ... Gather-scatter internal points ...
17
18 call MPI_Waitall(recv_requests)
19 call unpack_kernel(gather_buf, recv_buf, recv_map)
20
21 call MPI_Waitall(send_requests)
22 call device_sync()
23
24 ! ... Scatter shared points in gather_buf to u ...
```

Listing 4.2: Code to pack data from the gather buffer into the send buffers on the GPU. Each thread is assigned a single buffer index. The code has been simplified by removing nonessential details.

```
__global__ void pack_kernel(
    double *gather_buf,
    double *send_buf,
    int *send_map,
    int n)
{
    int j = threadIdx.x + blockDim.x*blockIdx.x;

    if (j >= n)
        return;

    send_buf[j] = gather_buf[send_map[j]-1];
}
```

Listing 4.3: Code to unpack data from the receive buffers into the gather buffer on the GPU. Each thread is assigned a single buffer index. The code has been simplified by removing nonessential details.

```
__global__ void unpack_kernel(
    double *gather_buf,
    double *recv_buf,
    int *recv_map,
    int n)
{
    int j = threadIdx.x + blockDim.x*blockIdx.x;

    if (j >= n)
        return;

    int idx = recv_map[j];
    double val = recv_buf[j];

    if (idx < 0) {
        atomicAdd(&gather_buf[-idx-1], val);
    } else {
        gather_buf[idx-1] += val;
    }
}
```

Chapter 5

Results and analysis

In this chapter, the results are presented and discussed. The result of the verification is presented in Section 5.1, and the performance results are presented in Section 5.2. The chapter ends with a discussion in Section 5.3.

5.1 Verification result

The verification runs showed that the baseline and accelerated implementations are numerically equivalent. The relative error in enstrophy as compared to the reference solution is shown in Figure 5.1. The figure shows that both the baseline implementation and the accelerated implementation reach a maximum of 6.7% relative error at $t = 14.4$. The error in enstrophy fluctuates during the run since the smallest scales are not resolved due to the coarser mesh.

5.2 Performance results

In this section, the results from the system performance tests, gather–scatter benchmark and the application benchmarks on Alvis are presented.

5.2.1 System performance

The latency of copies between host and device memory is shown in Figure 5.2. Up to 4 KB a copy between device and host memory takes 5.5 to 7.5 µs. Above 4 KB the latency increases roughly linearly.

The results of the message passing benchmark are shown in Figure 5.3. For inter-node communication, shown in Figure 5.3a, using host buffers or device

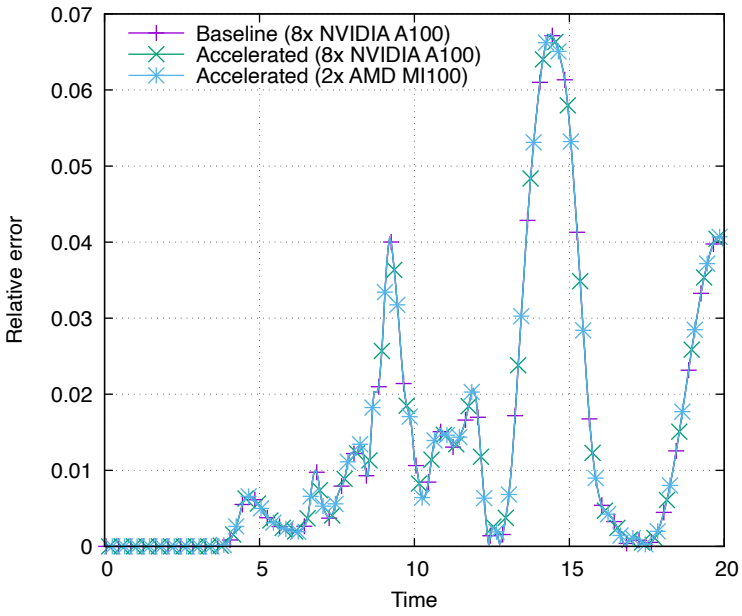


Figure 5.1: Relative error in enstrophy as compared to reference data in the verification case. The three lines overlap perfectly.

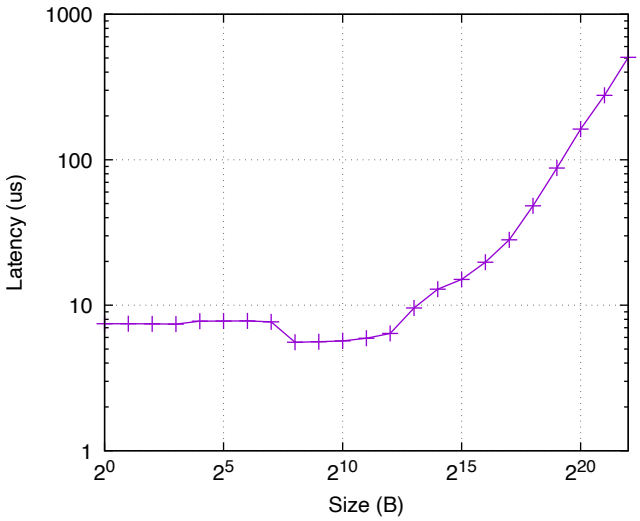


Figure 5.2: Time to complete a copy between host and device memory (logarithmic).

buffers leads to comparable performance, although host buffers are faster for most of the message size range. However, for intra-node communication, shown in Figure 5.3b, there is a significant difference between using host buffers and device buffers. There is a large spike in latency centered around message size 8 KB. In fact, it takes longer to send a message of size 8 KB than a message of size 256 KB. In contrast, for messages above 1 MB, the cost of using host buffers increases significantly while device buffers are fast.

5.2.2 Gather–scatter performance

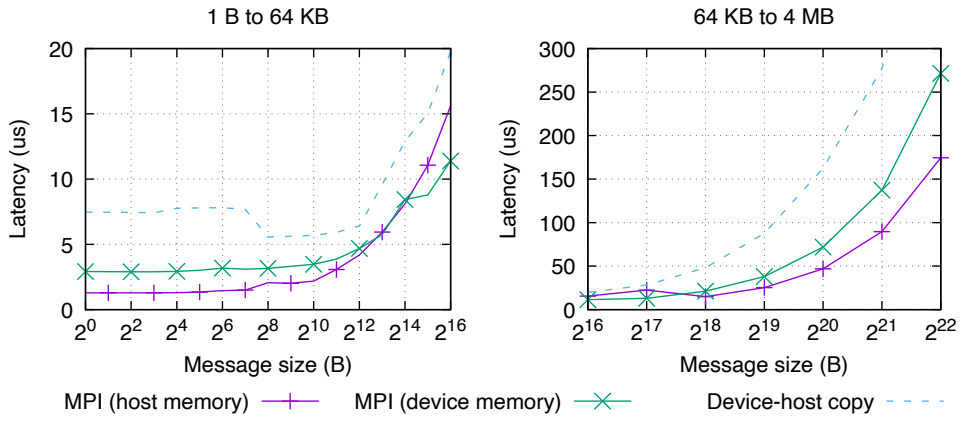
The performance of the new accelerated gather–scatter implementation as compared to the baseline implementation is shown in Figure 5.4. At polynomial orders 7 and 11 (Figure 5.4a), corresponding to the main solver, there is a significant speedup across the whole strong scaling range. For $N = 7$ the speedup is between 1.34 and 3.59. For $N = 11$ the speedup is between 1.23 and 3.65.

At polynomial orders 1 and 3 (Figure 5.4b), corresponding to the coarse grid preconditioner, the results are more varied. For $N = 1$ the results are better at 4 GPUs and similar at 64 and 128 GPUs. However, the performance is worse for 8 to 64 GPUs, with as much as 6.73 times slower for 32 GPUs. For $N = 3$ there is a 1.98 to 4.06 speedup between 4 to 32 GPUs, but with a similar spike in execution time at 64 GPUs where the operation takes 2.88 times as long.

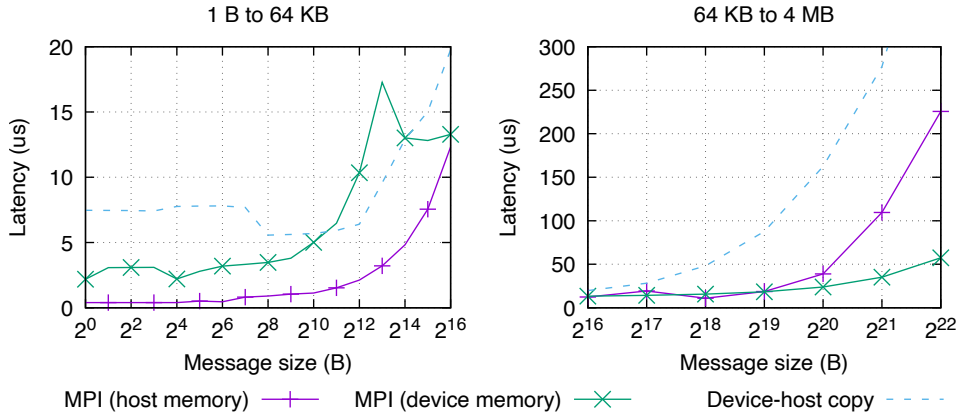
5.2.3 Application performance

The results of the application performance on the small case are shown in Figure 5.5, and on the large case in Figure 5.6. The results show that the performance of the application is improved by using the accelerated gather–scatter operation in both cases. For the small case, the speedup is between 1.08 and 1.66 depending on the number of GPUs used. For the large case, the speedup is between 1.58 and 2.59.

The scaling profile shown by the parallel efficiency is better for the accelerated version. At 128 GPUs the relative scaling performance is similar for both versions. However, since the accelerated version is faster at 16 GPUs, it is still also faster at 128 GPUs.

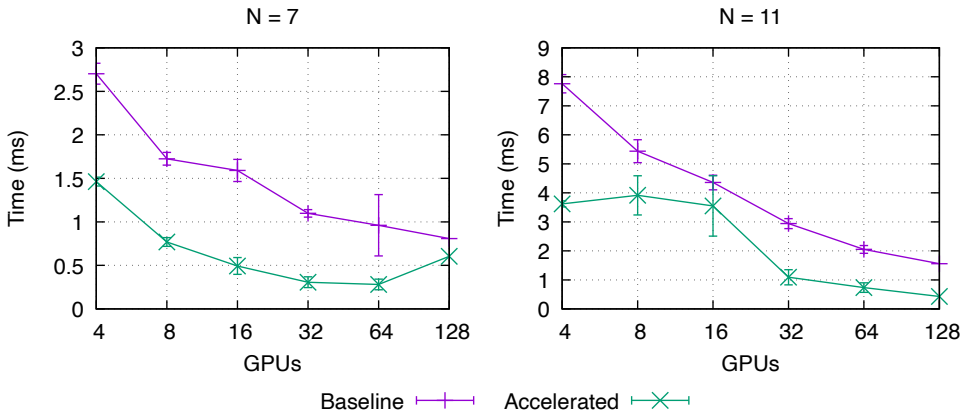


(a) Inter-node communication (between two nodes).

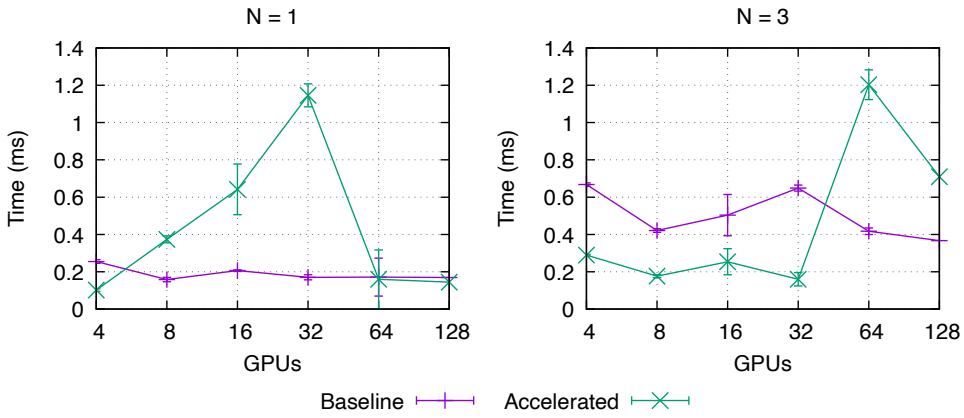


(b) Intra-node communication (within a single node).

Figure 5.3: Pingpong MPI latency, comparing messages in host memory and device memory. Time for copy between device and host memory is also shown as a reference.

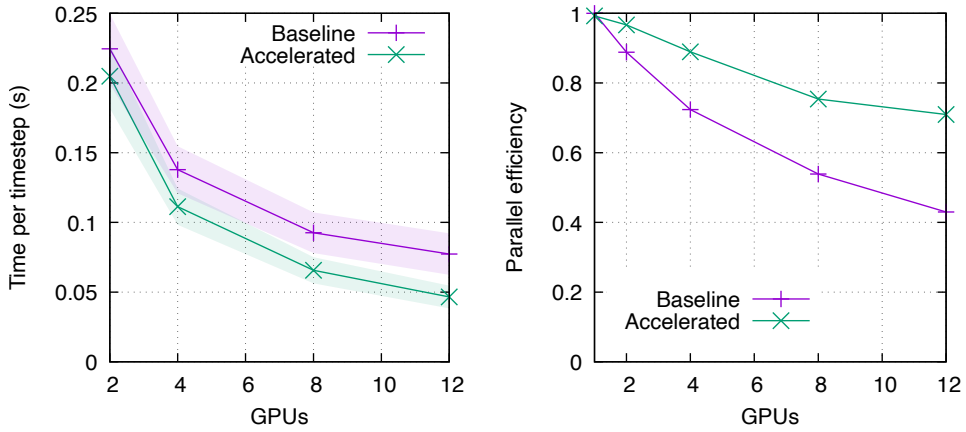


(a) Polynomial orders 7 and 11, corresponding to the main solver.



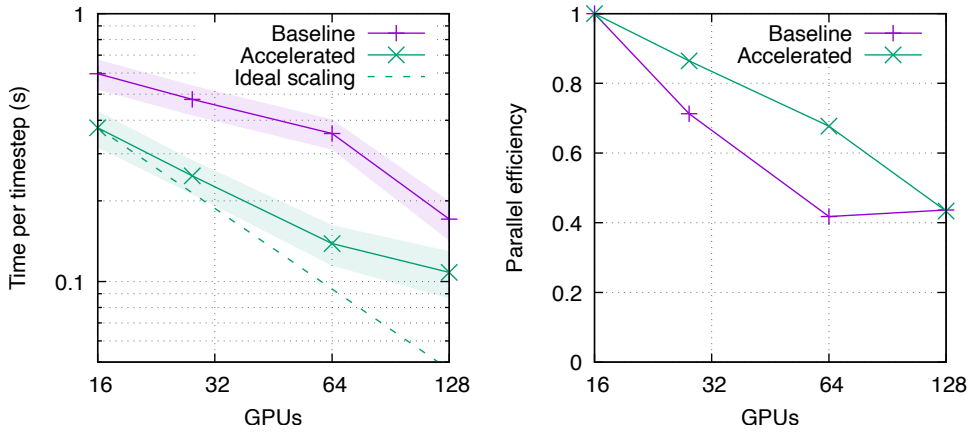
(b) Polynomial orders 1 and 3, corresponding to the coarse grid preconditioner.

Figure 5.4: Results of the gather–scatter benchmark, showing the average time to complete a gather–scatter operation. Note the different scales on the y-axes.



(a) Strong scaling (b) Parallel efficiency

Figure 5.5: Application performance on the small Taylor–Green vortex case. The shaded region indicates the standard deviation.



(a) Strong scaling (logarithmic) (b) Parallel efficiency

Figure 5.6: Application performance on the large Taylor–Green vortex case. The shaded region indicates the standard deviation.

5.3 Discussion

The results of the message passing benchmark show that the GPU-aware MPI implementation on the test system is not necessarily tuned perfectly for intra-node communication. The fact that it takes longer to send smaller messages than larger messages indicates that different transport strategies are used based on the message size. It is also surprising that transfer between device and host memory is slower than inter-node communication.

In the UCX library, which is used by the MPI library on Alvis, the transport strategy is selected dynamically. The UCX library can be configured using hundreds of options, shown with the `ucx_info -c` command. Future investigations could utilize the `UCX_PROTO_INFO=y` option to debug which strategies are selected by UCX [46], to see if it correlates with the observed MPI performance.

The inconsistent message passing performance is reflected in the results of the gather–scatter benchmark. In certain configurations, the lower order operations are significantly slower, which correlates with the spike in intra-node MPI latency. However, the communication pattern in the latency test and the gather–scatter operations are quite different. In the latency test, there are only two ranks sending one message at a time, while in the gather–scatter operation there are many ranks, sending several messages concurrently. Therefore the results are not directly comparable. Further investigation is necessary to find out why the accelerated gather–scatter operation is significantly slower in these specific configurations. It is also unclear if these findings generalize to other clusters, MPI implementations, etc.

However, for most configurations, the accelerated gather–scatter operation provides a significant speedup. The speedup may be explained by the fact that the baseline implementation first has to copy the gather buffer from device memory to host memory, and then copy it back after the communication is complete. As illustrated by Figure 5.3, the cost of copying memory between host and device is in most cases larger than MPI communication itself.

The results of the application performance evaluation show that the performance of the gather–scatter operation has a large impact on the overall application performance. Performance is improved for both test cases, but the improvement is larger for the larger case. This reinforces the fact that communication latency is key in strong scaling application performance. It also encourages further exploration of GPU-aware message passing in scientific computing.

Since each run was done in a separate job, they were executed on different

sets of compute nodes. To increase the reliability of the study, all runs could be executed on the same set of compute nodes.

Since the gather–scatter performance varies based on configuration, it may be beneficial to try different implementations at runtime and choose the fastest one dynamically. This is an approach taken by NekRS [28].

Chapter 6

Conclusions and future work

In this chapter, the conclusions of the degree project and how the work may be continued are described. The conclusions are presented in Section 6.1, limitations of the work are described in Section 6.2, future work is suggested in Section 6.3, and reflections on environmental aspects of the work are presented in Section 6.4.

6.1 Conclusions

Leveraging GPU-aware message passing to avoid copying data between host and device memory can significantly improve large-scale fluid simulations on GPU-accelerated supercomputers. However, performance may vary for different parts of the application, and therefore system-specific tuning may be needed. The answers to the research questions are:

1. On the test system, GPU-aware message passing with device memory buffers had a similar performance to conventional message passing with host memory buffers for most messages. However, in the range 1-64 KB the performance was significantly worse for intra-node communication, and above 1 MB the performance was significantly better.

2. By utilizing GPU-aware message passing in the gather–scatter operation, expensive data transfers between host and device memory can be avoided by packing and unpacking communication buffers on the GPU. It is possible to achieve up to a 2.59 speedup when running a fluid simulation across 64 GPUs using this method.

6.2 Limitations

All performance measurements have been done on a single NVIDIA A100 cluster. Performance characteristics may differ depending on the GPU model, interconnect type, etc. In particular, the upcoming AMD MI250X architecture is specifically optimized for network communication from device memory. The characteristics may also differ for other simulation cases with different domain sizes.

6.3 Future work

More work is needed to fully understand the observed inconsistent MPI performance. Runtime profiling and debug options may be used to show which transport strategy is used by the MPI library. Running the tests on other systems may reveal if the issues are related to the specific system, library, or something else.

To ensure the application is performance portable across systems, automatic selection of communication strategy can be implemented.

Since using MPI with device buffers has been successful for the gather–scatter operation it may be extended to other parts of the application as well. In particular, the other sections with MPI communication is a global vector sum operation and MPI-IO calls for writing data to disk.

Other communication models such as Partitioned Global Address Space using the NVSHMEM library may also be tried.

6.4 Reflections

Optimizations of high-performance software lead to reduced energy consumption. This means both that the software becomes more environmentally friendly, and that it becomes cheaper to use and thus enables new applications in science and engineering. In particular, fluid simulations are used to design more energy-efficient means of transport.

References

[1] TOP500.org, *Top500 list june 2022*, 2022. [Online]. Available: https://www.top500.org/lists/top500/2022/06/ (visited on 06/03/2022).

[2] NVIDIA, *CUDA C++ programming guide*, Version 11.7.0. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 05/12/2022).

[3] R. S. Rogallo and P. Moin, "Numerical simulation of turbulent flows", *Annual Review of Fluid Mechanics*, vol. 16, no. 1, pp. 99–137, Jan. 1984. DOI: 10.1146/annurev.fl.16.010184.000531.

[4] G. N. Coleman and R. D. Sandberg, "A primer on direct numerical simulation of turbulence – methods, procedures and guidelines", 2010. [Online]. Available: https://www.mech.kth.se/flow-old/nordita2010/coleman_dns.pdf (visited on 05/12/2022).

[5] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations", in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, ser. SC '99, New York, NY, USA: Association for Computing Machinery, Jan. 1, 1999, 68–es, ISBN: 9781581130911. DOI: 10.1145/331532.331599.

[6] J. Bashor. "ACM Gordon Bell Prize recognizes top accomplishments in running science apps on HPC". (Aug. 25, 2016), [Online]. Available: http://sc16.supercomputing.org/2016/08/25/acm-gordon-bell-prize-recognizes-top-accomplishments-running-science-apps-hpc/ (visited on 01/28/2022).

[7] N. Offermans, O. Marin, M. Schanen, *et al.*, "On the strong scaling of the spectral element solver nek5000 on petascale systems", in *Proceedings of the Exascale Applications and Software Conference 2016*, ACM, Apr. 2016. DOI: 10.1145/2938615.2938617.

[8] N. Jansson, M. Karp, A. Podobas, S. Markidis, and P. Schlatter, "Neko: A modern, portable, and scalable framework for high-fidelity computational fluid dynamics", *CoRR*, vol. abs/2107.01243, 2021. arXiv: 2107.01243.

[9] Á. Tanarro, R. Vinuesa, and P. Schlatter, "Effect of adverse pressure gradients on turbulent wing boundary layers", *Journal of Fluid Mechanics*, vol. 883, A8, 2020. DOI: 10.1017/jfm.2019.838.

[10] TOP500.org, *Green500 june 2022*, 2022. [Online]. Available: https://www.top500.org/lists/green500/2022/06/ (visited on 06/03/2022).

[11] T. Sterling, M. Anderson, and M. Brodowicz, *High performance computing: modern systems and practices*. Elsevier, 2018, ISBN: 0-12-420215-2. DOI: 10.1016/C2013-0-09704-6.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM Press, 1967. DOI: 10.1145/1465482.1465560.

[13] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters", *Computer Science - Research and Development*, vol. 26, no. 3, p. 257, Apr. 12, 2011, ISSN: 1865-2042. DOI: 10.1007/s00450-011-0171-3.

[14] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs", in *2013 42nd International Conference on Parallel Processing*, ISSN: 2332-5690, Oct. 2013, pp. 80–89. DOI: 10.1109/ICPP.2013.17.

[15] AMD, *AMD CDNA 2 architecture*, 2021. [Online]. Available: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf (visited on 01/18/2022).

[16] Oak Ridge National Laboratory, *Frontier*. [Online]. Available: https://www.olcf.ornl.gov/frontier/ (visited on 05/03/2022).

[17] PDC Center for High Performance Computing, *About Dardel*. [Online]. Available: https://www.pdc.kth.se/hpc-services/computing-systems/about-dardel-1.1053338 (visited on 05/03/2022).

[18] NVIDIA, *NVIDIA CUDA compute unified device architecture programming guide*, Version 1.0, 2007. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf (visited on 05/12/2022).

[19] AMD, *AMD Instinct MI100 instruction set architecture: Reference guide*, 2020. [Online]. Available: https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf (visited on 02/21/2022).

[20] AMD, *AMD Instinct high performance computing (HPC) and tuning guide*, Rev. 1027, 2021.

[21] J. Tu, G. H. Yeoh, and C. Liu, *Computational Fluid Dynamics: A Practical Approach*, eng. Saint Louis: Elsevier Science & Technology, 2018, ISBN: 9780081011270.

[22] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow* (Cambridge Monographs on Applied and Computational Mathematics). Cambridge: Cambridge University Press, 2002, ISBN: 9780521453097. DOI: 10.1017/CBO9780511546792.

[23] L. F. Richardson, *Weather Prediction by Numerical Process* (Cambridge mathematical library), eng, Second edition.. Cambridge: University Press, 2007, ISBN: 9780511340123.

[24] Clay Mathematics Institute, *Navier–stokes equation*. [Online]. Available: http://www.claymath.org/millennium-problems/navier%E2%80%93stokes-equation (visited on 01/25/2022).

[25] J. H. Spurk and N. Aksel, *Fluid Mechanics*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-30259-7.

[26] A. T. Patera, "A spectral element method for fluid dynamics: Laminar flow in a channel expansion", *Journal of Computational Physics*, vol. 54, no. 3, pp. 468–488, Jun. 1984. DOI: 10.1016/0021-9991(84)90128-1.

[27] G. E. Karniadakis, M. Israeli, and S. A. Orszag, "High-order splitting methods for the incompressible Navier-Stokes equations", eng, *Journal of computational physics*, vol. 97, no. 2, pp. 414–443, 1991, ISSN: 0021-9991. DOI: 10.1016/0021-9991(91)90007-8.

[28] P. Fischer, S. Kerkemeier, M. Min, *et al.*, "NekRS, a GPU-Accelerated Spectral Element Navier-Stokes Solver", *arXiv:2104.05829 [cs]*, Apr. 2021, arXiv: 2104.05829.

[29] R. Barrett, B. MW, T. Chan, *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Jan. 1, 1994, vol. 43. DOI: 10.1137/1.9781611971538.

[30] P. F. Fischer, "An overlapping schwarz method for spectral element solution of the incompressible navier–stokes equations", *Journal of Computational Physics*, vol. 133, no. 1, pp. 84–101, May 1, 1997, ISSN: 0021-9991. DOI: 10.1006/jcph.1997.5651.

[31] P. F. Fischer and J. W. Lottes, "Hybrid schwarz-multigrid methods for the spectral element method: Extensions to navier-stokes", in *Domain Decomposition Methods in Science and Engineering*, T. J. Barth, M. Griebel, D. E. Keyes, *et al.*, Eds., ser. Lecture Notes in Computational Science and Engineering, Berlin, Heidelberg: Springer, 2005, pp. 35–49, ISBN: 9783540268253. DOI: 10.1007/3-540-26825-1_3.

[32] P. F. Fischer, "Projection techniques for iterative solution of ax = b with successive right-hand sides", *Computer Methods in Applied Mechanics and Engineering*, vol. 163, no. 1, pp. 193–204, Sep. 21, 1998, ISSN: 0045-7825. DOI: 10.1016/S0045-7825(98)00012-7.

[33] N. Jansson, "Spectral element simulations on the NEC SX-aurora TSUBASA", in *The International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2021, New York, NY, USA: Association for Computing Machinery, Jan. 20, 2021, pp. 32–39, ISBN: 9781450388429. DOI: 10.1145/3432261.3432265.

[34] S. Markidis, J. Gong, M. Schliephake, *et al.*, "OpenACC acceleration of the nek5000 spectral element code", *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 311–319, Mar. 2015. DOI: 10.1177/1094342015576846.

[35] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton, "HipBone: A performance-portable GPU-accelerated c++ version of the NekBone benchmark", *arXiv:2202.12477 [cs]*, Feb. 24, 2022. arXiv: 2202.12477.

[36] N. Dryden, N. Maruyama, T. Moon, *et al.*, "Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems", in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, Nov. 2018, pp. 1–13. DOI: 10.1109/MLHPC.2018.8638639.

[37] R. T. Mills, M. F. Adams, S. Balay, *et al.*, "Toward performance-portable PETSc for GPU-based exascale systems", *Parallel Computing*, vol. 108, p. 102 831, Dec. 1, 2021, ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102831.

[38] J. Zhang, J. Brown, S. Balay, *et al.*, "The PetscSF scalable communication layer", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, Apr. 2022. DOI: 10.1109/tpds.2021.3084070.

[39] Z. Wang, K. Fidkowski, R. Abgrall, *et al.*, "High-order CFD methods: Current status and perspective", *International Journal for Numerical Methods in Fluids*, vol. 72, no. 8, pp. 811–845, 2013, ISSN: 1097-0363. DOI: 10.1002/fld.3767.

[40] Network-Based Computing Laboratory, *OSU micro-benchmarks*. [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/ (visited on 04/26/2022).

[41] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation", *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct. 2014, ISSN: 1558-2183. DOI: 10.1109/TPDS.2013.222.

[42] The Open MPI Project, *FAQ: Running CUDA-aware Open MPI*. [Online]. Available: https://www.open-mpi.org/faq/?category=runcuda (visited on 03/30/2022).

[43] AMD, *ROCm collective communications library*. [Online]. Available: https://github.com/ROCmSoftwarePlatform/rccl (visited on 04/26/2022).

[44] NVIDIA, *NVSHMEM*. [Online]. Available: https://developer.nvidia.com/nvshmem (visited on 05/09/2022).

[45] AMD, *ROCm OpenSHMEM (ROC_SHMEM)*. [Online]. Available: https://github.com/ROCm-Developer-Tools/ROC_SHMEM (visited on 04/26/2022).

[46] Unified Communication Framework, *OpenUCX frequently asked questions*. [Online]. Available: https://openucx.readthedocs.io/en/master/faq.html (visited on 05/09/2022).

TRITA-EECS-EX- 2022:00

For DIVA

{
"Author1": { "Last name": "Wahlgren",
"First name": "Jacob",
"Local User Id": "u1gn1jgz",
"E-mail": "jacobwah@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Degree1": {"Educational program": "Master's Programme, Computer Science, 120 credits"
,"programcode": "TCSCM"
,"Degree": "Master's Programme, Computer Science, 120 credits"
,"subjectArea": "Computer Science"
},
"Title": {
"Main title": "Using GPU-aware message passing to accelerate high-fidelity fluid simulations",
"Language": "eng" },
"Alternative title": {
"Main title": "Användning av grafikprocessormedveten meddelandeförmedling för att accelerera nogranna strömningsmekaniska datorsimuleringar",
"Language": "swe"
},
"Supervisor1": { "Last name": "Jansson",
"First name": "Niclas",
"Local User Id": "u1fr0htl",
"E-mail": "njansson@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "PDC Center for High Performance Computing" }
},
"Supervisor2": { "Last name": "Karp",
"First name": "Martin",
"Local User Id": "u1g1zh7w",
"E-mail": "makarp@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"Examiner1": { "Last name": "Markidis",
"First name": "Stefano",
"Local User Id": "u1y5heyt",
"E-mail": "markidis@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"National Subject Categories": "10206, 20306",
"Other information": {"Year": "2022", "Number of pages": "xv,46"},
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": "Hilaire Bouaddi"},
"Presentation": { "Date": "2022-05-27 10:00"
,"Language":"eng"
,"Room": "Room 4423, Ingrid Melinder"
,Äddress": Lindstedtsvägen 5, floor 4"
,"City": "Stockholm" },
"Number of lang instances": "2",

"Abstract[eng]": €€€€
Motivated by the end of Moore's law, graphics processing units (GPUs) are replacing general-purpose processors as the main source of computational power in emerging supercomputing architectures. A challenge in systems with GPU accelerators is the cost of transferring data between the host memory and the GPU device memory. On supercomputers, the standard for communication between compute nodes is called Message Passing Interface (MPI). Recently, many MPI implementations support using GPU device memory directly as communication buffers, known as GPU-aware MPI.
One of the most computationally demanding applications on supercomputers is high-fidelity simulations of turbulent fluid flow. Improved performance in high-fidelity fluid simulations can enable cases that are intractable today, such as a complete aircraft in flight.
In this thesis, we compare the MPI performance with host memory and GPU device memory, and demonstrate how GPU-aware MPI can be used to accelerate high-fidelity incompressible fluid simulations in the spectral element code Neko. On a test system with NVIDIA A100 GPUs, we find that MPI performance is similar using host memory and device memory, except for intra-node messages in the range of 1-64 KB which is significantly slower using device memory, and above 1 MB which is faster using device memory. We also find that the performance of high-fidelity simulations in Neko can be improved by up to 2.59 times by using GPU-aware MPI in the gather–scatter operation, which avoids several transfers between host and device memory. €€€€,

"Keywords[eng]": €€€€
high-performance computing, computational fluid dynamics, spectral element method, graphical processing units, message passing interface €€€€,

"Abstract[swe]": €€€€
Motiverat av slutet av Moores lag så har grafikprocessorer (GPU:er) börjat ersätta konventionella processorer som den huvudsakliga källan till beräkningingskraft i superdatorer. En utmaning i system med GPU-acceleratorer är kostnaden att överföra data mellan värdminnet och acceleratorminnet. På superdatorer är Message Passing Interface (MPI) en standard för kommunikation mellan beräkningsnoder. Nyligen stödjer många MPI-implementationer direkt användning av acceleratorminne som kommunikationsbuffertar, vilket kallas GPU-aware MPI.
En av de mest beräkningsintensiva applikationerna på superdatorer är nogranna datorsimuleringar av turbulenta flöden. Förbättrad prestanda i nogranna flödesberäkningar kan möjliggöra fall som idag är omöjliga, till exempel ett helt flygplan i luften.
I detta examensarbete jämför vi MPI-prestandan med värdminne och acceleratorminne, och demonstrerar hur GPU-aware MPI kan användas för

att accelerera nogranna datorsimuleringar av inkompressibla flöden i spektralelementkoden Neko. På ett testsystem med NVIDIA A100 GPU:er finner vi att MPI-prestandan är liknande med värdminne och acceleratorminne. Detta gäller dock inte för meddelanden inom samma beräkningsnod i intervallet 1-64 KB vilka är betydligt långsammare med acceleratorminne, och över 1 MB vilka är betydligt snabbare med acceleratorminne. Vi finner också att prestandan av nogranna datorsimuleringar i Neko kan förbättras upp till 2,59 gånger genom användning av GPU-aware MPI i den så kallade gather–scatter-operationen, vilket undviker flera överföringar mellan värdminne och acceleratorminne. €€€€,
"Keywords[swe]": €€€€
högprestandaberäkningar, beräkningsströmningsdynamik, spektralelementmetoden, grafikprocessorer, meddelandeförmedlingsgränssnitt €€€€,
}