CPU- and GPU-initiated Communication Strategies for Conjugate Gradient Methods on Large GPU Clusters

James D. Trotter
james@simula.no
Simula Research Laboratory
Oslo, Norway

Sinan Ekmekçibaşı
sekmekcibasi23@ku.edu.tr
Koç University
Istanbul, Turkey

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

Johannes Langguth
langguth@simula.no
Simula Research Laboratory
Oslo, Norway
University of Bergen
Bergen, Norway

Xing Cai
xingca@uio.no
University of Oslo
Oslo, Norway
Simula Research Laboratory
Oslo, Norway

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

Abstract

The Conjugate Gradient (CG) method is a key building block in numerous applications, yet its low computational intensity and sensitivity to communication overhead make it difficult to scale efficiently on multi-GPU systems. In light of recent advances in multi-GPU communication technologies, we revisit CG parallelization for large-scale GPU clusters.

This work presents scalable CG and pipelined CG solvers targeting NVIDIA and AMD GPUs, using GPU-aware MPI, NCCL/RCCL and NVSHMEM to implement both CPU- and GPU-initiated communication schemes. We also introduce a monolithic variant that offloads the entire CG loop to the GPU, enabling fully device-initiated execution via NVSHMEM. Optimizations across all variants reduce unnecessary data transfers and synchronization overheads; the GPU-initiated variant eliminates CPU involvement altogether.

We benchmark our implementations on NVIDIA- and AMD-based supercomputers using SuiteSparse matrices and a real-world finite element application. By avoiding data transfers and synchronization bottlenecks, our single-GPU implementations achieve 8–14 % performance gains over state-of-the-art solvers. In strong scaling tests on over 1,000 GPUs, we outperform existing approaches by 5–15 %.

While CPU-initiated variants remain favorable due to a lack of vendor supported device-side computational kernels and suboptimal NVSHMEM configurations at the clusters, the strong scaling properties of the GPU-initiated CG variant indicates that it will be highly competitive at even larger GPU counts and with further tuning.

CCS Concepts

• **Mathematics of computing → Mathematical software**; • **Computing methodologies → Parallel algorithms**; **Distributed algorithms**.

Keywords

GPU, Conjugate Gradient, CUDA, HIP, GPU-aware MPI, NCCL, RCCL, NVSHMEM

1 Introduction

The Conjugate Gradient (CG) method [27] is a widely used iterative solver for large, sparse, symmetric and positive-definite linear systems, forming a critical building block in many scientific and industrial applications. Its core operations, such as vector additions, dot products, and sparse matrix-vector multiplications, are all low in computational intensity, making CG highly sensitive to communication and synchronization overhead and challenging to scale on multi-GPU systems where communication bandwidth lags behind GPU compute performance.

Recent advancements in communication technologies have introduced a range of options for multi-GPU programming [18, 53]. Programmers can leverage GPU-aware MPI, NCCL/RCCL, and NVSHMEM/rocSHMEM, all supporting direct GPU-to-GPU communication initiated from the CPU. Further enhancements, such as GPUDirect RDMA [43], enable low-latency transfers, while device-initiated communication [25] reduces CPU involvement and synchronization overheads. Given these developments, it is time to revisit multi-GPU implementations of CG. The aim is to conduct comparative studies that can provide practical guidance on communication library choices and highlight key trade-offs in implementing this critical solver for the HPC community.

This paper presents scalable multi-GPU CG solvers targeting both NVIDIA and AMD platforms. We implement and evaluate variants using CPU- and GPU-initiated communication via GPU-aware MPI, NCCL, RCCL, and NVSHMEM. Each variant supports both CG and pipelined CG algorithms [21], with CUDA and HIP implementations for NVIDIA and AMD GPUs, respectively. To provide a context for our multi-GPU strategies, we begin with a baseline single-GPU CG implementation, emphasizing our improvements to reduce CPU-GPU synchronization. In our multi-GPU designs, communication is initiated either from the host CPU (MPI, NCCL, or RCCL) or directly on the GPU (via NVSHMEM).

The CPU-initiated approach aligns with typical multi-GPU linear solvers, which are based on GPU-aware MPI. In addition, we also present a novel adoption of the NCCL/RCCL collective communication libraries and GPU streams for CPU-initiated communication in multi-GPU CG. The purpose is to avoid communication-related synchronisation points between CPU and GPU and thus improve scalability.

Going even further, we develop a new, *monolithic* version of multi-GPU CG, where the entire CG loop is offloaded to the GPU in a single kernel. Moreover, we propose a scalable, one-sided communication scheme for the underlying irregular point-to-point communication, where NVSHMEM is used to initiate the communication directly from the GPU kernel, without involving the CPU.

We conduct a rigorous performance evaluation on three supercomputers: LUMI, MareNostrum 5, and Wisteria. Our assessment uses matrices from SuiteSparse [17], and augments these with matrices for large-scale tests, including a case from 3D cardiac modelling simulations on a realistic heart mesh.

Our paper makes the following contributions:

- By carefully avoiding GPU-to-CPU data transfer and synchronisation, we improve the performance of CG on a single GPU by up to 8 % on AMD MI250X, 14 % and 11 % on NVIDIA A100 and H100, respectively.
- The single-GPU improvements are shown to carry over to typical multi-GPU implementations of CG with GPU-aware MPI, outperforming a state-of-the-art numerical library by 7–14 % on 1024 AMD MI250X GPUs and 256 NVIDIA H100 GPUs, the largest number that we had access to for this work.
- We compare the performance of CG (and pipelined CG) when GPU-aware MPI and NCCL/RCCL are used for internode GPU-to-GPU communication, showing that NCCL is highly mature and often competitive with GPU-aware MPI, especially when collective Allreduce operations dominate the performance. In contrast, RCCL significantly underperforms, primarily due to inefficient Allreduce operations for small message sizes.
- We develop a scalable, monolithic CG implementation with GPU-initiated communication that shows strong scaling properties equal to or better than CG with CPU-initiated communication (on 256 GPUs). We believe this will be the best alternative at even larger scales, after further tuning of the underlying SpMV kernel and NVSHMEM configuration.

2 Background

2.1 Multi-GPU communication libraries

To communicate between GPUs, there are multiple libraries available from both GPU vendors and research groups. In this section, we discuss the most widely used ones.

2.1.1 GPU-aware MPI. Due to the widespread use of MPI in the HPC community, early GPU communication libraries extended MPI to support efficient GPU-to-GPU communication, eliminating unnecessary device-to-host copies. This capability, known as GPU-aware MPI, allows passing GPU memory pointers directly to MPI functions, enabling efficient data transfers via GPUDirect RDMA for NVIDIA GPUs or ROCm RDMA for AMD GPUs [2, 43]. GPU-aware MPI is supported by several implementations, including OpenMPI, MPICH, and MVAPICH2 [28, 47, 48, 51, 54–56]. However, a limitation of GPU-aware MPI is its lack of awareness of GPU-specific constructs, such as *streams*, which are critical for overlapping computation and communication. Recent efforts have aimed to bridge this gap by integrating GPU streams and other GPU features into MPI [24, 26, 41, 42].

2.1.2 Collective Communication Libraries. Multiple GPU vendors offer topology-aware collective primitives for inter-GPU communication that we term as GPU Collective Communication Libraries (GPU-CCL). These libraries have been designed by vendors to be natively compatible with their own GPU programming model via pipelining operations to a GPU stream with their host-side API. Due to their performance and built-in topology awareness, GPU-CCLs have been integrated into several state-of-the-art deep learning frameworks, including Pytorch and Tensorflow [57] as a communication backend. Prominent examples include NCCL (NVIDIA), RCCL (AMD), and oneCCL (Intel) [6, 31, 45].

2.1.3 One-sided GPU Communication. For point-to-point communication, the libraries discussed above adopt a two-sided communication model, where both sender and receiver actively participate in the data exchange. In contrast, one-sided communication allows only one party, either the sender or the receiver, to initiate and complete the transfer, while the other side remains passive. This model is implemented by Partitioned Global Address Space (PGAS) libraries such as OpenSHMEM and was introduced into the MPI standard starting from version 2.0.

Recently GPU vendors have provided implementations of the OpenSHMEM specification for GPUs, specifically NVIDIA's NVSHMEM [44], AMD's rocSHMEM [3] and Intel's SHMEM [30]. These GPU-based SHMEM libraries extend the OpenSHMEM API specification, by providing stream-aware host-side APIs and device-side APIs, to perform communication at a granularity involving single GPU threads, warps or thread blocks.

2.1.4 GPU-initiated communication. Recent developments in interconnect technologies from GPU vendors have allowed support for the InfiniBand verbs interface on GPUs. The GPU SHMEM libraries are thus able to issue inter-node communication from the GPU directly to the NIC, bypassing the CPU entirely [1, 46, 49, 50]. Unlike GPU-aware MPI and NCCL/RCCL, where GPU-to-GPU communication is still issued from the CPU, this allows true GPU-initiated communication which can reduce the need for CPU-GPU synchronization.

In this paper, we focus on GPU-initiated communication with NVSHMEM, which is enabled by one of two underlying communication methods, known as IBRC and IBGDA (Infiniband GPUDirect Async) [49]. Only the latter can be used in a mode where the GPU is truly autonomous by explicitly disabling the use of a CPU proxy thread. Otherwise, the proxy thread is used in IBGDA by notifying the NIC and GPU at various times during communication. However, we found that the system configurations of large GPU clusters, such as BSC's *MareNostrum 5* and CINECA's *Leonardo*, do not currently support IBGDA. A smaller cluster, *Wisteria*, which is used in our experiments in Section 4, does support IBGDA, but only in the CPU proxy thread mode. Ultimately, we resorted to IBRC also in this case, because IBGDA showed worse performance for collective communication with small message sizes.

We expect that AMD's rocSHMEM can be used for similar one-sided, GPU-initiated communication, but it is not supported on *LUMI*, the AMD-based system used in our experiments (Section 4.1).

2.2 Conjugate gradient methods

Solving large systems of linear algebraic equations, in the form of $Ax = b$, is a central component in numerous applications of computational science. The system matrix A is often ill-conditioned and sparse (with very few nonzeros), so iterative linear solvers constitute the most suitable strategy. The *conjugate gradient* (CG) algorithm [27] is the best iterative solver when A is symmetric and positive-definite. CG is also important because its three computational kernels: vector addition (AXPY), dot-product (DOT) and sparse matrix-vector multiplication (SpMV), are the building blocks of other powerful iterative solvers that are based on Krylov subspaces.

Each iteration of the original CG algorithm consists of three AXPYs, two DOTs and one SpMV. All these kernels have very low computational intensity, making parallel implementations of CG very sensitive to the communication overhead. Scaling CG across many GPUs is notoriously difficult due to the imbalance between high GPU compute performance and relatively slow inter-GPU communication.

We also include a *pipelined* CG variant [21] in this study. The pipelined version aims to reduce the number of global synchronizations (e.g., due to MPI_Allreduce) at the cost of additional computations. We note that there are many variants of pipelined CG [12–14] and s-step CG [9–11]. However, our goal is not to provide a comprehensive survey, but rather to focus on two widely used CG variants and evaluate their parallel scalability in the context of modern multi-GPU programming.

3 Implementation

This section describes in detail our multi-GPU CG implementations, which are based on CPU- and GPU-initiated communication. The CPU-initiated variants include implementations using GPU-aware MPI, NCCL, and RCCL, while the GPU-initiated variant is built on NVSHMEM. Each of these variants is implemented for both the standard and pipelined CG algorithms. To ease the discussion, we first review a typical single-GPU version with an emphasis on CPU-GPU synchronisation and data transfers.

3.1 Single-GPU implementation of CG

A typical GPU implementation of CG (see Listing 1) offloads AXPY (lines 10, 11 & 16), DOT (lines 3, 7 & 12) and SpMV (lines 2 & 6) to the GPU. The other operations, such as computing the scalars α and β (lines 9 and 15) and testing for convergence (line 14), are usually performed on the host CPU. This approach is the basis for the CG implementation in the widely used numerical library PETSc [40].

Optimised DOT and AXPY kernels are available from vendor BLAS libraries, i.e., cuBLAS [16] and hipBLAS [4]. Moreover, cuSPARSE [15] and hipSPARSE [5] provide optimised SpMV kernels for common storage formats, such as compressed sparse row (CSR).

```
1   Input: $A$, $b$, $x_0$, $\epsilon$
2   GPU SpMV: $r_0 \leftarrow b - A * x_0$
3   GPU DOT:  $\rho_0 \leftarrow r_0 \cdot r_0$; copy $\rho_0$ from GPU to CPU
4   CPU SYNC: Wait until $\rho_0$ is copied from GPU
5   GPU COPY: $s \leftarrow r_0$
6   For $k = 1, 2, \ldots$ Do
7       GPU SpMV: $t \leftarrow A * s$
8       GPU DOT:  $\gamma \leftarrow s \cdot t$; copy $\gamma$ from GPU to CPU
9   (*) CPU SYNC: Wait until $\gamma$ is copied from GPU
10  (*) CPU:      $\alpha \leftarrow \rho_{k-1}/\gamma$
11  (*) GPU AXPY: $x_k \leftarrow \alpha s + x_{k-1}$
12      GPU AXPY: $r_k \leftarrow -\alpha t + r_{k-1}$
13      GPU DOT:  $\rho_k \leftarrow r_k \cdot r_k$; copy $\rho_k$ from GPU to CPU
14      CPU SYNC: Wait until $\rho_k$ is copied from GPU
15      CPU:      If $\sqrt{\rho_k} \le \epsilon\sqrt{\rho_0}$ Then Exit (Converged)
16  (*) CPU:      $\beta \leftarrow \rho_k/\rho_{k-1}$
17  (*) GPU AXPY: $s \leftarrow \beta s + r_k$
18  End For
```

Listing 1: Pseudo-code for a typical single-GPU CG, e.g., as implemented by PETSc. Lines marked with (*) are modified in our version of CG (see Section 3.1.2).

The CSR SpMV kernel provided by cuSPARSE is based on the merge-based SpMV algorithm [38], which has better load-balancing properties compared to a naive implementation.

3.1.1 Synchronisation. GPU kernels in CUDA/HIP execute asynchronously, so explicit synchronisation is needed in Listing 1 before the CPU can proceed with operations that depend on results produced by the GPU. Specifically, synchronisation is needed after the DOTs that compute γ (line 8) and ρ_k (line 13).

3.1.2 Our improvements. The synchronisation point associated with γ (line 8) can be eliminated, if γ is retained in the GPU device memory throughout the computation. Instead of returning the result of the DOT kernel to the CPU, which implies a device-to-host copy, we use `cublasSetPointerMode` in cuBLAS (and similarly for hipBLAS) so that γ remains in the GPU device memory. However, doing so forces all scalar coefficients to reside in the GPU device memory, particularly those used as inputs to the AXPY kernels (e.g., α and β). We propose to fuse the computation of α and β with the relevant AXPY kernels, rather than launching separate kernels to compute those single scalar values.

We thus obtain a single-GPU implementation of CG, where only a single synchronisation point is needed (line 13), i.e., prior to the convergence test performed by the host. Furthermore, only a single scalar value, ρ_k, which represents the square of the (implicit) residual norm, must be transferred to the host in each iteration. Here, we delay the AXPY kernel that updates the solution vector (line 11) until after the DOT kernel on line 13 and use GPU streams to asynchronously overlap it with the copy of ρ_k to the host.

3.2 Multi-GPU parallelization

3.2.1 Partitioning. To use CG in a distributed-memory parallel setting with multiple GPUs, one must first distribute the work and the associated data among the GPUs. A scalable algorithm must 1) distribute the matrix A, right-hand side b, and solution vector x,

2) balance the computational load between GPUs, and 3) minimise parallel overhead due to communication and synchronisation. We follow common practice (e.g., PETSc [7], Hypre [20]) by distributing the matrix rowwise and partitioning the vectors accordingly. Specifically, we use the METIS graph partitioner [33] to distribute the matrix rows, which promotes load balance through constraints and reduces communication volume by minimising the *edge cut* of the unweighted, undirected graph corresponding to the symmetric, sparse matrix A.

3.2.2 Communication. Two kinds of communication are needed in the multi-GPU CG algorithm. First, since the vectors are distributed, every dot product must be followed by a collective communication in the form of an *allreduce* to sum up the partial results from all GPUs. Second, SpMV operations require an additional communication step before they can be carried out on each GPU.

Consider an SpMV $y = A * x$ and a rowwise matrix partitioning. Each GPU possesses a partial matrix $A^{(p)}$ and is responsible for computing a partial output vector $y^{(p)}$, both of which contain values only from the rows that they were assigned. On any GPU, a given value of x is only needed if the corresponding column of its partial sparse matrix $A^{(p)}$ is nonzero. As a result, the typical communication pattern that arises in connection with SpMV is an *irregular point-to-point communication* that depends on the sparsity pattern of the underlying matrix.

In light of these communication requirements, a common convention is to divide a matrix $A^{(p)}$ into two parts, $A^{(p)} = \bar{A}^{(p)} + \hat{A}^{(p)}$, and then perform a distributed SpMV in three steps:

(1) Multiply the local part $\bar{A}^{(p)}$ of the matrix with the local part of x that is readily available without the need for communication.
(2) Perform point-to-point communication to receive $\hat{x}$, denoting values of x needed to perform the remaining SpMV.
(3) Multiply the remaining part $\hat{A}^{(p)}$ of the matrix with $\hat{x}$, using the values received during the point-to-point communication.

This method furthermore provides a possibility to overlap the communication in step 2 with the SpMV computation in step 1, potentially hiding the communication cost. The overlapping strategy is used in PETSc and in our own CG implementations, as explained in further detail below.

3.3 CG with host-initiated communication

The usual programming model with multiple GPUs requires the host CPU to retain control over execution and initiate communication (in addition to launching GPU kernels and managing CPU-GPU data transfers). In this subsection, we discuss our implementation of distributed-memory parallel CG using host-initiated communication with GPU-aware MPI or NCCL/RCCL. A pseudo-code is shown in Listing 2. Partial matrices and vectors are denoted with a superscript, e.g., $\boldsymbol{A}^{(p)}$ and $\boldsymbol{b}^{(p)}$, but we omit the superscript for the auxiliary vectors (i.e., $\boldsymbol{r}$, $\boldsymbol{s}$ and $\boldsymbol{t}$) for the sake of readability.

SpMV, AXPY and DOT kernels are now performed by GPUs on their local matrices and vectors. The scalars α, β and γ remain in GPU memory throughout, and the computation of the former two is fused with AXPY kernels (lines 17, 21 and 24). CPUs launch all

```
 1     Input: $A^{(p)} = \bar{A}^{(p)} + \hat{A}^{(p)}$, $b^{(p)}$, $x_0^{(p)}$, $\epsilon$
 2         GPU SpMV:     $r_0 \leftarrow b^{(p)} - A^{(p)} * x_0^{(p)}$
 3         GPU DOT:      $\rho_0^{(p)} \leftarrow r_0 \cdot r_0$
 4     (+) CPU SYNC:     Wait until $\rho_0^{(p)}$ is ready on GPU
 5         ALLREDUCE:    $\rho_0 \leftarrow \sum_p \rho_0^{(p)}$
 6         GPU COPY:     $s \leftarrow r_0$
 7         For $k = 1, 2, \ldots$ Do
 8             GPU PACK:     Pack $\hat{s}$ for sending
 9     (+)     CPU SYNC:     Wait until $\hat{s}$ is ready on GPU
10             P2P COMM:     Start sending $\hat{s}$ to remote GPUs
11             GPU SpMV:     $t \leftarrow \bar{A}^{(p)} * s$
12             P2P SYNC:     Wait until $\hat{s}$ is received
13             GPU SpMV:     $t \leftarrow t + \hat{A}^{(p)} * \hat{s}$
14             GPU DOT:      $\gamma^{(p)} \leftarrow s \cdot t$
15     (+)     CPU SYNC:     Wait until $\gamma^{(p)}$ is ready on GPU
16             ALLREDUCE:    $\gamma \leftarrow \sum_p \gamma^{(p)}$
17             GPU AXPY:     $\alpha \leftarrow \rho_{k-1}/\gamma$;  $r_k \leftarrow -\alpha t + r_{k-1}$
18             GPU DOT:      $\rho_k^{(p)} \leftarrow r_k \cdot r_k$
19     (+)     CPU SYNC:     Wait until $\rho_k^{(p)}$ is ready on GPU
20             ALLREDUCE:    $\rho_k \leftarrow \sum_p \rho_k^{(p)}$; start copying $\rho_k$ to CPU
21             GPU AXPY:     $\alpha \leftarrow \rho_{k-1}/\gamma$;  $x_k^{(p)} \leftarrow \alpha s + x_{k-1}^{(p)}$
22             CPU SYNC:     Wait until $\rho_k$ is copied from GPU
23             CPU:          If $\sqrt{\rho_k} \le \epsilon\sqrt{\rho_0}$ Then Exit (Converged)
24             GPU AXPY:     $\beta \leftarrow \rho_k/\rho_{k-1}$;  $s \leftarrow \beta s + r_k$
25         End For
```

Listing 2: Our multi-GPU implementation of CG with host-initiated communication. Some synchronisation points are mandatory for GPU-aware MPI, but they are not needed for stream-aware GPU communication (e.g., NCCL/RCCL). These are indicated by lines marked with (+).

kernels asynchronously, initiate asynchronous transfers of ρ_k from GPUs (line 20), and synchronise with their GPU on line 22, before carrying out the convergence test.

Finally, host CPUs initiate all communication between GPUs. Each DOT kernel is immediately followed by a collective allreduce operation (lines 5, 16 and 20). Point-to-point communication associated with SpMV (lines 8–12) starts with i) packing messages on the GPU, followed by ii) posting (non-blocking) message sends and receives from the CPU, and iii) waiting for messages to be received. The local part of the SpMV is carried out in the meantime to overlap with the communication.

In any case, the data itself travels directly between peer GPU memories, if they are connected directly (e.g., NVLink or Infinity Fabric), or between GPU and NIC, provided that the underlying communication library supports it.

3.3.1 GPU-aware MPI. In the case of MPI, the necessary collective operations are performed with `MPI_Allreduce`. Point-to-point communication uses persistent, non-blocking send/receive, later followed by `MPI_Waitall` to ensure completion. Persistent APIs reduce overhead [34] compared to standard `MPI_Isend`/`MPI_Irecv` by instead calling `MPI_Isend_init` and `MPI_Irecv_init` only once during setup, and then invoking `MPI_Startall` for each point-to-point exchange. We assume that the MPI library is GPU-aware, so that pointers to GPU memory are passed directly to MPI calls, and that it supports asynchronous progress, so that point-to-point communication overlaps with SpMV computation.

Nevertheless, MPI communication imposes a need for additional synchronisation points, because the host CPU must wait for data produced by the GPU before it can be used in the appropriate MPI calls. This limitation arises from the discrepancy between the MPI programming model and the asynchronous execution model that pervades GPU programming. Specifically, in Listing 2, three additional synchronisation points are needed in each CG iteration. In practice, the host CPU calls, e.g., `cudaDeviceSynchronize`, on lines 9, 15 and 19, to ensure prior GPU kernels complete before calling `MPI_Isend` (or `MPI_Startall`) (line 10) and `MPI_Allreduce` (lines 16 and 20).

3.3.2 NCCL/RCCL. GPU collective communication libraries offer direct equivalents for most MPI calls. In particular, we use `ncclAllReduce` to perform allreduce operations for both NCCL and RCCL. (Note that the libraries share identical APIs.) Non-blocking point-to-point communication is achieved by enclosing a group of calls to `ncclSend` and `ncclRecv` with calls to `ncclGroupStart` and `ncclGroupEnd`.

Furthermore, NCCL and RCCL avoid the problem of excessive CPU-GPU synchronisation, because communication can be queued asynchronously by the host CPU onto GPU streams. Communication is thus carried out by the GPU as soon as previously queued kernels are completed. There is no need for the CPU to wait until the GPU is finished, and then have the GPU wait idly for the CPU again until the communication is initiated. In Listing 2, the SpMV, AXPY and DOT kernels are executed in the default stream, whereas message packing (line 8), `ncclSend` and `ncclRecv` (line 10) are executed in a secondary, non-blocking stream, allowing it to overlap with local SpMV (line 11). The synchronisation on line 12 is achieved by asynchronously recording an event (`cudaEventRecord` or `hipEventRecord`) in the secondary stream and making the default stream wait until the event has been reached.

3.4 CG with GPU-initiated communication

In this subsection, we present a novel implementation of CG that departs from the traditional model of offloading a sequence of individual kernels to the GPU. This work improves the CPU-free, autonomous execution model proposed by Ismayilov et al. [32]. Our motivation stems from the fact that coordination between CPU and GPU inevitably incurs overheads related to kernel launches, CPU-GPU synchronisation and data transfers. Here we pose the question: Can these overheads be avoided by offloading the *entire CG algorithm* to the GPU in a single, monolithic kernel?

Our implementation relies on two recent developments in GPU programming. The first is *cooperative groups* (since CUDA 9/ROCm 5), a feature that allows grid-wide synchronisation within CUDA/HIP kernels. The term *grid* refers to all of the GPU threads executing a kernel together on the GPU. Previously, the primary method of achieving synchronisation among all threads in a grid was to end the kernel and then have the CPU launch a new one. With cooperative groups, one can instead insert synchronisation points directly into the GPU kernel code to ensure that all threads (or a subset of threads) reach the same point before proceeding. This is needed, for instance, to make the result of a dot product available to all GPU threads in a grid for use in subsequent computations.

```
Input: $A^{(p)} = \bar{A}^{(p)} + \hat{A}^{(p)}$, $\hat{A}^{(p)}$, $b^{(p)}$, $x_0^{(p)}$, $\epsilon$
GRID SpMV:      $r_0 \leftarrow b^{(p)} - A^{(p)} * x_0^{(p)}$
GRID DOT:       $\rho_0^{(p)} \leftarrow r_0 \cdot r_0$
GRID SYNC:      Wait for all GPU threads
ALLREDUCE:      $\rho_0 \leftarrow \sum_p \rho_0^{(p)}$
GRID SYNC:      Wait for all GPU threads
GRID COPY:      $s \leftarrow r_0$
For $k = 1, 2, \ldots$ Do
  THREAD 0:     $\alpha \leftarrow \rho_{k-1}^{(p)}$;  $\beta \leftarrow \rho_{k-1}^{(p)}$
  GRID SYNC:    Wait for all GPU threads
  THREAD 0:     $\gamma \leftarrow 0$;  $\rho_k^{(p)} \leftarrow 0$
  GRID COPY:    $t \leftarrow 0$
  GRID SYNC:    Wait for all GPU threads
  GRID PACK:    Pack $\hat{s}$ for sending to remote GPU
  P2P WAIT:     Wait for ready signal from remote GPUs
  GRID SYNC:    Wait for all GPU threads
  P2P SIGNAL:   Put-with-signal $\hat{s}$ to remote GPUs
  GRID SpMV:    $t \leftarrow \bar{A}^{(p)} * s$
  P2P WAIT:     Wait for signal that $\hat{s}$ is received
  P2P SIGNAL:   Unpack message to $\hat{s}$; signal to remote GPUs
  GRID SYNC:    Wait for all GPU threads
  GRID SpMV:    $t \leftarrow t + \hat{A}^{(p)} * \hat{s}$
  GRID SYNC:    Wait for all GPU threads
  GRID DOT:     $\gamma^{(p)} \leftarrow s \cdot t$
  GRID SYNC:    Wait for all GPU threads
  ALLREDUCE:    $\gamma \leftarrow \sum_p \gamma^{(p)}$
  GRID SYNC:    Wait for all GPU threads
  GRID AXPY:    $\alpha \leftarrow \rho_{k-1}/\gamma$;  $x_k^{(p)} \leftarrow \alpha s + x_{k-1}^{(p)}$
  GRID AXPY:    $\alpha \leftarrow \rho_{k-1}/\gamma$;  $r_k \leftarrow -\alpha t + r_{k-1}$
  GRID DOT:     $\rho_k^{(p)} \leftarrow r_k \cdot r_k$
  GRID SYNC:    Wait for all GPU threads
  ALLREDUCE:    $\rho_k \leftarrow \sum_p \rho_k^{(p)}$
  GRID SYNC:    Wait for all GPU threads
  GRID:         If $\sqrt{\rho_k} \le \epsilon\sqrt{\rho_0}$ Then Exit (Converged)
  GRID AXPY:    $\beta \leftarrow \rho_k/\rho_{k-1}$;  $s \leftarrow \beta s + r_k$
End For
```

Listing 3: Our monolithic, multi-GPU implementation of CG with GPU-initiated communication.

Second, with the support of GPU-initiated communication, the point-to-point and collective communications needed in multi-GPU CG can be invoked from within the GPU kernel code itself. For this, we use NVIDIA's NVSHMEM, which offers one-sided, GPU-initiated communication primitives. AMD's counterpart, rocSHMEM, can be substituted and used, but is unfortunately not supported on LUMI, the AMD-based system used in our experiments (Section 4.1).

A pseudo-code for our monolithic CG kernel is shown in Listing 3. Most operations are performed jointly by all threads in the grid (indicated with GRID), while some are carried out by a single thread (THREAD 0). Point-to-point communication (described in detail below) also involves operations at the granularity of a thread block.

3.4.1 BLAS operations. Vendor-optimised BLAS kernels (e.g., cuSPARSE/hipSPARSE) are generally not available to be used from within a GPU kernel, and we therefore provide our own versions. AXPY is implemented as a grid-stride loop, whereas DOT is a grid-stride loop over the vector entries, followed by a tree-based, warp-level reduction and atomic operations to sum contributions from all warps. A subsequent grid-wide synchronisation is needed to ensure that all warps have finished. Finally, we implement SpMV using the merge-based algorithm [38], which is also used by cuSPARSE.

3.4.2 Communication. For allreduce, we use NVSHMEM's built-in collectives (`nvshmem_double_sum_reduce`), called from a single GPU thread. Although NVSHMEM allows a full warp or thread block to join the communication, we saw no performance benefit in the case of communicating only a single scalar value.

We implement one-sided point-to-point communication by sending each message in a *non-blocking put-with-signal* operation, i.e., `nvshmemx_double_put_signal_nbi_block` is called by an entire thread block, which is recommended to achieve higher throughput, especially for communication over NVLink [44]. Signals are used as a mechanism by the sender to indicate when it has finished depositing its message in the recipient's buffer. They are also used by the recipient to notify the sender (`nvshmem_signal_op`) when the message has been unpacked from the recipient's buffer, which can then be reused for the next message. GPUs typically receive messages from multiple remote GPUs, and different messages are handled by different thread blocks and its own pair of signals. In lines 15 and 19 in Listing 3, a single thread in the block waits for the signal (`nvshmem_signal_wait_until`) and a block-wide synchronisation is performed before all threads in the block proceed.

3.4.3 Synchronisation. While the monolithic approach completely eliminates any form of CPU-GPU synchronisation, there are still a number of GPU-wide synchronisation points present. It is prudent to keep these to a minimum, as they are expensive, but cannot be avoided when work is distributed across multiple thread blocks and data dependencies between thread blocks arise. This is clearly the case for dot products and allreduce communication, but it also occurs due to scalars α, β, γ and $\rho_k^{(p)}$ being shared by all threads. Grid synchronisation is also needed specifically due to using the merge-based SpMV algorithm, because it assigns threads to entries of the input/output vectors in a different way than the surrounding BLAS operations.

3.4.4 Programming challenges. While programming distributed-memory GPU codes already presents many difficulties, using a one-sided communication model, such as NVSHMEM, poses additional challenges. The programmer must be even more aware of synchronisation and manage it explicitly to ensure that remote data is read or written at the right time. More specifically related to GPU-initiated communication, NVSHMEM generally offers three APIs for the same operation at different levels of granularity. The programmer must decide if a communication primitive, e.g., a put or allreduce operation, should be performed by a single GPU thread, or a group of threads in the form of a warp or a thread block. Such choices impact synchronisation that may need to be performed by the GPU threads, and it is often unclear what implications such choices have for intranode (e.g., NVLink) and internode (e.g., Infiniband) communication performance.

3.5 Pipelined CG

Our implementations of multi-GPU pipelined CG with CPU- and GPU-initiated communication both follow the same principles as outlined in the previous two sections. The main difference is that the two allreduce collective communications are instead combined into a single allreduce operation in pipelined CG. In turn, pipelined CG performs six AXPY operations per iteration instead of three, thus requiring more FLOPs. It also requires three additional vectors in the working memory, which can negatively impact data locality. Nonetheless, an overall benefit is expected when the number of GPUs is very large and allreduce becomes a dominating factor. This version of pipelined CG is known as Pipelined Chronopoulos/Gear CG, shown as Algorithm 3 in Ghysels and Vanroose [21].

4 Experiments

We evaluate the performance of our proposed CG and pipelined CG implementations on three supercomputers with NVIDIA and AMD GPUs. In these experiments, we refer to our CG implementations as *aCG*. The versions with CPU-initiated communication (Section 3.3) based on GPU-aware MPI, NCCL and RCCL are termed *aCG (MPI)*, *aCG (NCCL)* and *aCG (RCCL)*, respectively, whereas the GPU-initiated communication version (Section 3.4) is referred to as *aCG (NVSHMEM)*. PETSc's CG and pipelined CG implementations, which are based on GPU-aware MPI, are used for comparison.

4.1 Experimental setup

4.1.1 Supercomputers. We use three different supercomputers for our experiments, whose technical specifications are described below. Note that we report all network bandwidths as unidirectional values.

LUMI is an AMD-based machine with 2 978 nodes in the LUMI-G partition. Each node features one 64-core AMD Trento EPYC 7A53 CPU and four AMD Instinct MI250X Accelerators, consisting of two Graphics Compute Dies (GCDs) each. The GCDs are accessed as individual GPUs, for a total of 8 GPUs per node. Each pair of GPUs is connected by one to four Infinity Fabric links with a bandwidth of 400 Gb/s each. The network is a Cray Slingshot 11 interconnect with Dragonfly topology, and each node has four HPE Cray Cassini-1 NICs with a bandwidth of 200 Gb/s each. The NICs are directly connected to the GPUs. On LUMI, we use ROCm 6.0.3 with RCCL 2.18.3 and Cray MPICH 8.1.29.

MareNostrum 5 has 1 120 nodes in the Accelerated (ACC) partition, each of which has four NVIDIA H100 GPUs with 64 GB of HBM2e and two Intel Xeon Platinum 8460Y+ 40-core CPUs. All GPU pairs are connected via NVLINK 4.0 with a bandwidth of 1.2 Tb/s. Each GPU is also connected to an NVIDIA ConnectX-7 NIC with 200 Gb/s. The network is NDR Infiniband with a three-layer fat-tree topology [8]. The system's policies limit jobs to 100 nodes (400 GPUs), of which we use up to 64 nodes (256 GPUs).

Wisteria is a smaller system whose Wisteria-A (Aquarius) partition consists of 32 nodes with tightly coupled NVIDIA GPUs. Each node has 8 NVIDIA A100-SXM4-40GB GPUs, two Intel Xeon Platinum 8360Y CPUs, and 4 Mellanox ConnectX-6 NICs with 100 Gb/s. The GPUs are arranged in an HGX configuration, which allows all pairs to communicate at 2.4 Tb/s simultaneously. The network is HDR Infiniband with a full bisection bandwidth fat-tree topology. The maximum number of nodes per job is 8 (64 GPUs).

4.1.2 Software. On *MareNostrum*, we use NVIDIA HPC SDK 25.1, CUDA 12.6 and HPC-X MPI 2.21, whereas on *Wisteria* we have NVIDIA HPC SDK 24.1, CUDA 12.3 and HPC-X MPI 2.17.1. On both machines, we use NCCL 2.18.5 and NVSHMEM 3.1.7 with the IBRC transport (see Section 2.1.4). The IBGDA transport is not supported on *MareNostrum* and we disabled it on *Wisteria* due to

Table 1: Sparse matrices used in experiments

Matrix	Rows	Nonzeros	Nonzeros/Row Min/Max/Mean
Queen_4147	4,147,110	316,548,962	24 / 81 / 76.3
Bump_2911	2,911,419	127,729,899	1 / 195 / 43.9
Cube_Coup_dt6	2,164,760	124,406,070	24 / 68 / 57.5
Flan_1565	1,564,794	114,165,372	24 / 81 / 73.0
audikw_1	943,695	77,651,847	21 / 345 / 82.3
Serena	1,391,349	64,131,971	15 / 249 / 46.1
poisson1d	1,073,741,824	3,221,225,470	2 / 3 / 3.0
heart15	874,424,445	7,185,411,541	5 / 77 / 8.2

lower allreduce performance compared to IBRC. PETSc 3.21.5 and METIS 5.1.0 are used on all systems.

The solver codes, including PETSc, are highly tuned through the same compiler flags for the host CPU (e.g., `-O3 -march=native`) and optimised for the GPU architecture (e.g., using NVCC compiler flags `-gencode=arch=compute_80,code=sm_80` for NVIDIA A100, and HIP compiler flags `–offload-arch=gfx90a` for MI250X). Regarding communication, we have ensured that collectives are optimised by using NVIDIA's HPC-X and HCOLL for optimised collectives on *MareNostrum* and *Wisteria*, and vendor-optimised Cray MPICH on *LUMI*. Furthermore, we have used the recommended tuning from [18].

4.1.3 Test matrices. The matrices used in this study are listed in Table 1. We selected the largest relevant ones, i.e., real-valued and symmetric, from SuiteSparse [17]. In addition, we created two larger matrices for the scaling experiments. The first is a tridiagonal matrix from discretising a 1D Poisson equation on a uniform grid, designed to stress the DOT and AXPY kernels and global reductions. The second is an irregular matrix from a 3D cardiac monodomain simulation on an unstructured, tetrahedral mesh based on a realistic geometry of the human heart [37]. The last matrix is representative of applying a linear finite element method on a real geometry.

4.1.4 Reported measurements. For each matrix, we create a manufactured solution vector x^* by generating uniformly random values in the range $[-1, 1]$ and then normalising, so that the 2-norm of x^* is equal to one. The corresponding right-hand side is computed by multiplying $b = Ax^*$, and the linear system is solved with a zero initial guess and a tolerance of 10^{-6} for the relative residual $\|b - Ax\|/\|b\|$.

Solver time is measured by the host CPU on each MPI process using `clock_gettime`. All solvers perform 10 warmup iterations followed by MPI barrier and CPU-GPU synchronisation (e.g., `cudaDeviceSynchronize`) prior to starting the timer. CPU-GPU synchronisation is also performed before stopping the timer to ensure GPU work completion. We then report the maximum time over all MPI processes involved. For a given matrix and GPU count, we use identical matrix partitioning and measure all solvers including PETSc within the same job submission, ensuring consistent partitioning, nodes, and GPUs for fair comparison. Each measurement is repeated three times, and the best result is reported to minimize the impact of outliers or transient network issues.

For our CG solvers with CPU-initiated communication, we also perform some additional measurements for detailed analysis, where individual GPU kernel times are measured through manual, fine-grained instrumentation and accurate CUDA/HIP event-based timing. The overhead of this instrumentation is generally below 5 %.

In addition to solver times, we calculate FLOP rates and memory throughput by counting SpMV, DOT, and AXPY operations along with their corresponding FLOPs and memory traffic. We distinguish DOT from NRM2, the latter being a dot product of a vector with itself. For a matrix with m rows and n nonzeros, the FLOP count is $2n$ for SpMV and $2m$ for DOT, NRM2 and AXPY, whereas memory traffic in bytes is $20(n + m)$ for SpMV, $24m$ for AXPY, $16m$ for DOT and $8m$ for NRM2. Values are assumed to be in double precision floating point, whereas indices are 4-byte integers. Moreover, k iterations of CG result in $k + 1$ SpMV, k DOT, $k + 1$ NRM2, and $3k$ AXPY operations. In comparison, k iterations of pipelined CG performs $k + 2$ SpMV, k DOT, k NRM2 and $6k$ AXPY operations.

4.2 SuiteSparse experiments

4.2.1 Single GPU performance. We first test the solvers on the matrices from SuiteSparse using a single GPU. Since there is no communication in this scenario, we refer to our single-GPU CG implementation from Section 3.3 as *aCG (standard)* and the single-GPU monolithic version from Section 3.4 as *aCG (monolithic)*.

Large variations in running time is expected between different matrices due to matrix size and the different number of iterations the solver takes to converge. However, for a given matrix, the iteration counts are roughly the same across solvers and machines, but may vary slightly due to round-off errors. As a result, we only show iteration counts from measurements on one of the machines (in this case *MareNostrum 5*).

Results are shown in Table 2. aCG (standard) consistently outperforms PETSc by approximately 5–10%, primarily due to the synchronization improvements discussed in Section 3.1.2. It is also about 5–10% faster than aCG (monolithic), mainly because the latter uses our own merge-based SpMV implementation. Recall that aCG (standard) and PETSc both benefit from cuSPARSE/hipSPARSE's optimized SpMV. An exception to this trend is Queen_4147 on AMD MI250X, where our merge-based SpMV outperforms the one from hipSPARSE on this particular matrix. Since aCG (monolithic) and PETSc provide roughly the same performance, we conclude that the monolithic approach indeed has less overhead, and the impact is comparable to the difference between our own and the vendor-optimised SpMV kernels.

We also list the FLOP rate in GFLOP/s, which is based on the total calculated number of FLOPs for all iterations, as described in Section 4.1.4 above. These values are fairly constant for each solver and GPU, except for Queen_4147 on MI250X.

More importantly, the attained memory bandwidth is close to the maximum of 1.6 TB/s (MI250X GCD), 1.56 TB/s (A100), and 2.02 TB/s (H100). Note that *MareNostrum 5* uses the 64 GB version of the H100, which has lower memory bandwidth than the full 80 GB SXM version. Due to the large caches of both NVIDIA GPUs of 40 MB (A100) and 30 MB (H100), calculated bandwidth exceeds the available memory bandwidth as some vectors are partially cached.

Table 2: Single GPU performance of aCG and PETSc on all three machines. TB/s refers to the calculated memory throughput. The single-GPU versions of our CG algorithms from Sections 3.3 and 3.4 are referred to as *aCG (standard)* and *aCG (monolithic)*.

Single GPU Performance			MI250X GCD (*LUMI*)			A100 (*Wisteria*)			H100 (*MareNostrum 5*)		
Matrix	Solver	Iterations	Time [s]	GFLOP/s	TB/s	Time [s]	GFLOP/s	TB/s	Time [s]	GFLOP/s	TB/s
Queen_4147	PETSc	26 418	251.86	73.28	0.74	99.54	186.52	1.88	85.80	215.69	2.18
	aCG (standard)	26 165	247.54	74.97	0.76	94.10	198.43	2.00	79.91	229.36	2.32
	aCG (monolithic)	26 642	153.95	117.26	1.18	102.47	180.41	1.82	87.32	213.72	2.16
Bump_2911	PETSc	25 035	53.26	133.17	1.35	43.67	167.73	1.70	36.40	195.72	1.99
	aCG (standard)	25 290	50.10	143.84	1.46	39.84	185.94	1.89	33.62	214.09	2.18
	aCG (monolithic)	26 342	63.04	115.79	1.18	42.88	174.45	1.77	37.44	200.20	2.03
Cube_Coup_dt6	PETSc	408	0.80	140.96	1.43	0.64	175.93	1.78	0.54	207.81	2.10
	aCG (standard)	408	0.75	150.50	1.52	0.60	188.75	1.91	0.51	221.56	2.24
	aCG (monolithic)	408	0.98	115.39	1.17	0.64	177.88	1.80	0.56	203.25	2.06
Flan_1565	PETSc	1 158	2.39	121.59	1.23	1.60	181.38	1.83	1.35	215.70	2.18
	aCG (standard)	1 155	2.27	127.91	1.29	1.46	198.04	2.00	1.26	229.33	2.32
	aCG (monolithic)	1 154	2.39	121.24	1.22	1.58	183.82	1.86	1.35	214.35	2.16
audikw_1	PETSc	4 753	7.11	110.99	1.12	4.32	183.89	1.86	3.67	213.21	2.15
	aCG (standard)	4 772	6.81	117.34	1.18	3.83	204.65	2.07	3.36	233.98	2.36
	aCG (monolithic)	4 941	6.78	117.44	1.19	4.41	183.18	1.85	3.82	213.05	2.15
Serena	PETSc	3 700	4.37	121.48	1.23	3.17	168.13	1.71	2.66	199.05	2.02
	aCG (standard)	3 703	4.03	132.20	1.34	2.77	191.95	1.95	2.39	221.69	2.25
	aCG (monolithic)	3 745	4.68	114.15	1.16	3.07	174.62	1.77	2.81	190.47	1.93

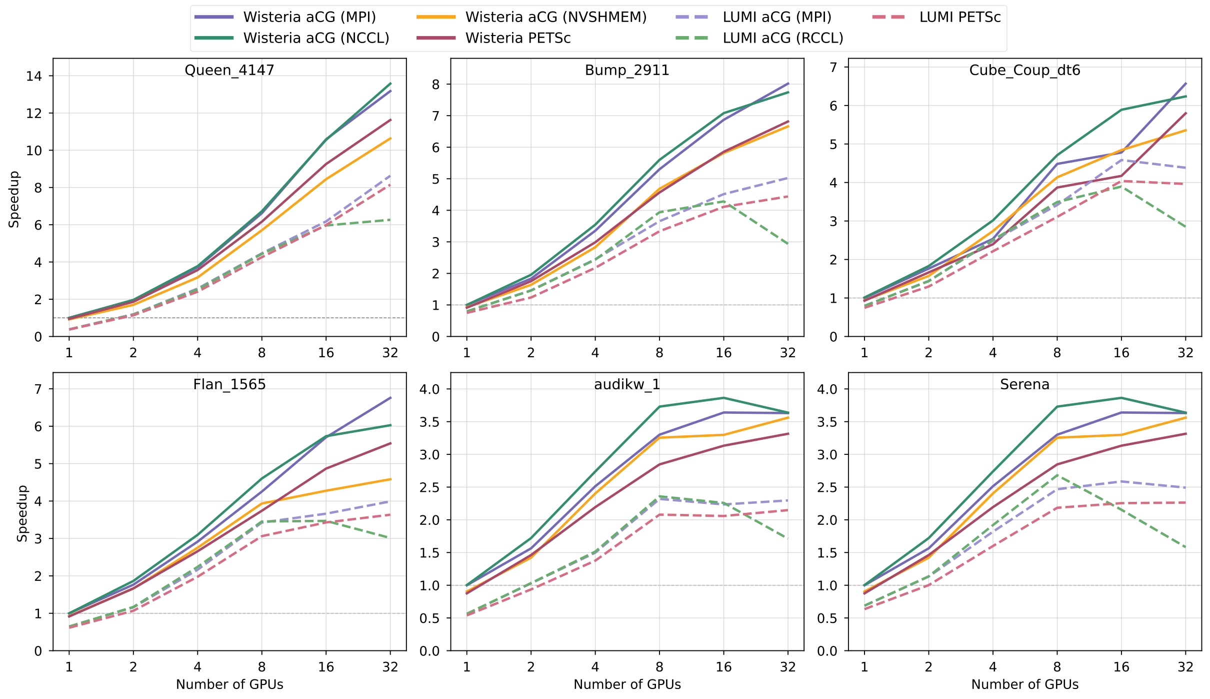

Figure 1: Speedups on SuiteSparse instances on *LUMI* and *Wisteria*. Value are relative to the running time of aCG (MPI) on one A100 GPU. A log scale is used on the horisontal axis.

Table 3: Performance per GPU for the large matrix instances of the base configuration of 64 GPUs on *LUMI* and 16 GPUs on *MareNostrum 5*. Time is given in seconds. For aCG with GPU-initiated communication, results are only shown on *MareNostrum 5*, since *LUMI* does not support rocSHMEM.

Performance per GPU		*LUMI* (64 GPUs; AMD MI250X GCD)						*MareNostrum 5* (16 GPUs; NVIDIA H100)					
		CG			Pipelined CG			CG			Pipelined CG		
	Solver	Time [s]	GFLOP/s	TB/s	Time [s]	GFLOP/s	TB/s	Time [s]	GFLOP/s	TB/s	Time [s]	GFLOP/s	TB/s
poisson1d	PETSc	2.86	87.43	0.96	4.78	71.88	0.81	8.81	113.34	1.25	15.76	87.16	0.98
	aCG (MPI)	2.20	113.64	1.25	2.77	123.96	1.40	6.83	146.29	1.61	8.35	164.54	1.86
	aCG (RCCL/NCCL)	2.49	100.23	1.10	2.90	118.41	1.33	6.82	146.59	1.61	8.33	164.98	1.86
	aCG (NVSHMEM)							8.00	124.91	1.37	9.53	144.38	1.63
heart15	PETSc	0.57	93.13	0.97	0.73	84.09	0.89	1.27	162.71	1.69	1.88	129.55	1.37
	aCG (MPI)	0.49	105.12	1.09	0.57	107.99	1.14	1.09	189.72	1.97	1.25	194.67	2.06
	aCG (RCCL/NCCL)	0.53	98.26	1.02	0.61	100.71	1.07	1.11	185.97	1.93	1.24	194.86	2.07
	aCG (NVSHMEM)							1.46	143.95	1.50	1.63	151.21	1.60

4.2.2 Scalability. Next, we study the scaling on the SuiteSparse matrices, using up to 32 GCDs/GPUs on *LUMI* and *Wisteria*, as most matrices are too small to benefit from larger GPU counts. Our versions of multi-GPU CG with CPU-initiated communication are derived from aCG (standard) and are referred to as *aCG (MPI)* or *aCG (RCCL/NCCL)*, depending on the underlying communication library that is used. The monolithic CG implementation is *aCG (NVSHMEM)*, and there is no counterpart for AMD GPUs due to lack of rocSHMEM support on *LUMI*.

Results are shown in Figure 1. On both machines, aCG (MPI) is consistently ahead of PETSc, though the difference for 32 GPUs is 10–22 % on Wisteria, but only 2–6 % on LUMI. Furthermore, NCCL performs on average 7 % better than aCG (MPI) for intranode communication on Wisteria (i.e., up to 8 GPUs), while the internode case of 16 and 32 GPUs is less clear and depends on the matrix. RCCL performs similar to aCG (MPI), but very poorly on 32 GPUs. More detailed profiling shows that the difference is due to allreduce, which is 2.3 to 3.0× faster when using MPI. aCG (NVSHMEM) lags behind the other solvers on the A100 but catches up for 32 GPUs on the small instances. This suggests that its main advantage, reduced synchronization latency, is present but not large enough to compete with the other solvers at the scale of 32 GPUs.

If we consider one of *LUMI's* full MI250X accelerators, which consists of 2 GCDs/GPUs, then we find that its performance is between 10 and 40 % higher than the performance of an NVIDIA A100 GPU (see Table 2). Thus, while detailed energy measurements are beyond the scope of this paper, the results indicate that the GPUs have a similar performance per watt since the MI250X has a 20% higher TDP.

4.2.3 Comparison to other works. Finally, we compare results from Table 2 and Figure 1 for our monolithic, NVHSMEM-based solver on NVIDIA A100 to results reported by Ismayilov et al. [32] and Ma et al. [35] concerning similar CG solvers, also based on NVSHMEM. We focus on the matrix Flan_1565, since it is used in all three studies. On a single A100 GPU, the performance is 397.5, 880 and 732.9 iterations per second for [32], [35] and our implementation, respectively. The differences are attributed to different SpMV kernels used in each case. The use of tensor cores in [35] provides about 20 % speedup over the vendor's cuSPARSE kernels used in our implementation.

On 8 A100s in a single node, connected with NVLink, the performance is 2458.5 and 3121.9 iterations per second for [32] and our implementation, respectively. The results are thus roughly comparable in the single-node scenario. We also performed additional experiments using the code provided by Ismayilov et al. [32], and found that performance drops dramatically to 2.0 iterations per second in a 2-node setup (16 GPUs) due to inefficient P2P communication. We expect the implementation by Ma et al. [35] to perform similarly, because it is based on the same P2P communication scheme. Ultimately, the communication-related improvements presented in this paper allow our monolithic, NVSHMEM-based solver to scale to multiple nodes and large GPU counts.

4.3 Large-scale experiments

We test the solvers as well as their pipelined versions on the large matrix instances, using up to 256 GPUs on *MareNostrum 5* and up to 1024 GPUs on *LUMI*. The job size limit of 64 GPUs on *Wisteria* prevents us from running large scale experiments on that machine. As before, aCG with GPU-initiated communication is only shown for NVSHMEM, as there is no support for rocSHMEM on *LUMI*.

4.3.1 Baseline performance. Table 3 shows the runtime and performance per GPU of the CG and pipelined CG solvers for 64 GPUs on *LUMI* and 16 GPUs on *MareNostrum 5*. For the matrix poisson1d, all solvers perform 930 iterations, whereas for heart15, the iteration counts range from 92 to 95. In any case, the FLOP rates and memory throughput reported for each solver factors out differences in iteration counts.

The fact that aCG (MPI) and aCG (RCCL/NCCL) outperform PETSc by 15–30 % highlights that the benefit of reducing synchronisation carries over from the single GPU case. Interestingly, the gap between MPI and RCCL is far smaller than it was for the SuiteSparse matrices in Figure 1, as allreduce is not yet dominating performance at this scale. PETSc and NVSHMEM are considerably slower, with the pipelined version from PETSc performing especially weak.

Note that the pipelined versions perform a larger number of operations to obtain the same result. Thus, their FLOP rates and

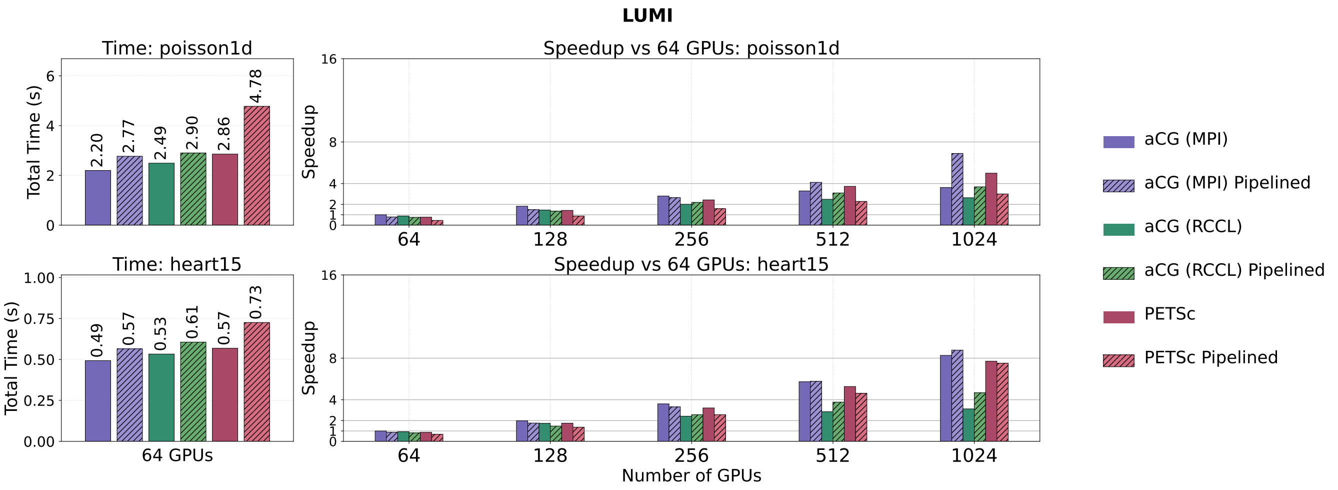

Figure 2: Running time and scaling of the large matrix instances on *LUMI*. Speedups are relative to the running time of aCG (MPI) on 64 GPUs. Log scales are used on the horisontal axis of the rightmost plots.

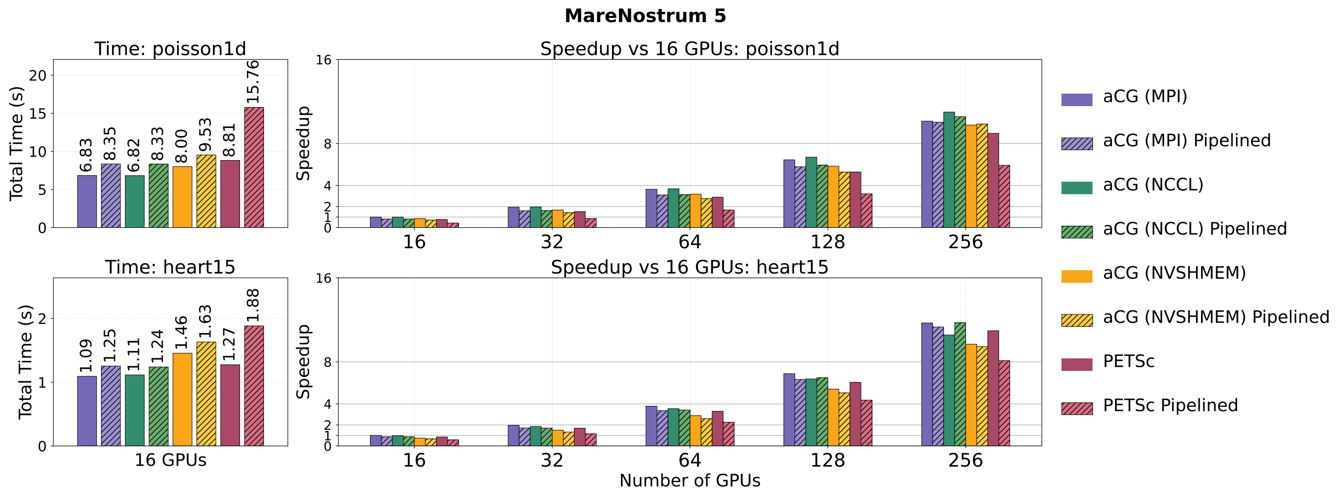

Figure 3: Runtime and scaling for the large matrices on *MareNostrum 5*. Speedups are relative to the running time of aCG (MPI) in heart15 and acg (NCCL) in poisson1d on 16 GPUs. Log scales are used on the horisontal axis of the rightmost plots.

attained bandwidths are higher, even though they take a longer time to solution. MPI and NCCL perform very similarly.

4.3.2 Scalability. Scaling to larger runs is shown in Figures 2 and 3 for *LUMI* and *MareNostrum 5*, respectively. As discussed above, we compare time to solution rather than FLOPs, using the time of aCG (MPI) in Table 3 as the basis for speedup.

For poisson1D on *LUMI*, we observe that the pipelined version of aCG (MPI) scales better. This is expected since the runtime of this instance is dominated by allreduce, and the pipelined versions only perform one such operation per time step. More detailed analysis shows that aCG (MPI) spends 53 % of the time in allreduce for 1024 GPUs. Furthermore, our measurements show that PETSc's CG outperforms aCG for very large GPU counts (512 or more). Further experiments confirm that it is in fact *better* to let the CPU perform the reduction on data located in host memory, which PETSc does, instead of passing a pointer to GPU memory when calling `MPI_Allreduce`. In this case, the lower latency of performing the reduction on data in host memory outweighs the additional CPU-GPU data copies and synchronisation that is needed.

For heart15 on *LUMI*, where running time is far more dependent on the point-to-point communication, the differences are much smaller. Both CG and pipelined CG scale well and attain similar performance for aCG (MPI) and PETSc. aCG (RCCL), on the other hand, scales poorly, which is in stark contrast to NCCL.

On *MareNostrum 5*, the NCCL versions perform best when allreduce is emphasised (poisson1d), whereas MPI performs best when point-to-point communication is most important (heart15). Specifically, for 256 GPUs, NCCL is about 5–10 % faster than MPI on poisson1D, whereas MPI is about 10 % faster than NCCL for heart15. The NVSHMEM versions are in any case catching up, and it is likely that for an even larger system, they would be able to outperform the alternatives. PETSc's pipelined CG again performs poorly, though further experiments beyond 256 GPUs are needed to conclude about its scalability. Apart from that, the difference between CG and pipelined CG is less pronounced on *MareNostrum 5*, indicating that allreduce is not a major bottleneck.

Overall, the codes show excellent strong scaling behavior with the best solvers maintaining a parallel efficiency of 76 % for 256 GPUs and 46 % for 1024 GPUs. This suggests that they are suitable for solving extremely large systems.

5 Related work

Communication is a bottleneck in large-scale systems and adoption of GPUs has focused the attention on GPU-to-GPU communication, as surveyed by Unat et al. [53] and De Sensi et al. [18]. The latter compares GPU-aware MPI to NCCL/RCCL for intra- and internode communication on large NVIDIA and AMD systems—Alps, Leonardo and LUMI—using microbenchmarks for pairwise point-to-point communication, all-to-all and allreduce collectives. They found that NCCL consistently outperforms GPU-aware MPI for allreduce on NVIDIA systems. Although RCCL does the same for large message sizes on LUMI, it only attains 10–25 % of the performance compared with GPU-aware MPI for allreduce on small messages (e.g., 8 to 64 B)—the most critical case for iterative solvers, such as CG. Our results indeed confirm these observations.

Hsu et al. [29] conducted an early evaluation of NVSHMEM's usability and its performance on Summit with matrix multiplication and a Jacobi solver as examples. For Jacobi, NVSHMEM shows comparable performance to MPI at least on up to 2 048 nodes (12 288 NVIDIA V100 GPUs). Groves et al. [22] compare NVSHMEM's CPU- and GPU-initiated internode communication on Summit, pointing out inefficiencies in NVSHMEM's GPU-initiated communication related to the use of a CPU progress thread. While InfiniBand GPUDirect Async (IBGDA) [49] seeks to address this, we found that system configurations on MareNostrum 5, Wisteria and other clusters we encountered, either did not allow for IBGDA at all or still required a CPU proxy thread anyway.

PETSc [7] is a well-known library for iterative solvers and serves as a baseline for comparison in our experiments. The communication layer used by PETSc solvers is known as PETScSF [58], and it relies on GPU-aware MPI. Mills et al. [39] identified and discussed the challenges of GPU communication in PETSc's iterative solvers, and suggested replacing MPI with NCCL and/or NVSHMEM to perform GPU stream-aware communication. Faibussowitsch et al. [19] later studied how to incorporate GPU streams in PETSc, and Mills et al. [40] recently adopted the host-side APIs of NVSHMEM for stream-aware, CPU-initiated communication in PETScSF. Furthermore, using stream-aware communication and skipping convergence tests in some iterations proved to reduce synchronisation overhead considerably.

Our monolithic kernel implementation of CG with GPU-initiated communication follows the persistent kernel paradigm [23] (also similar to the concept of megakernel or ubershader in graphics [52]). This is enforced when using CUDA/HIP cooperative groups for in-kernel synchronisation, because all GPU threads in a thread block must be scheduled concurrently on a Streaming Multiprocessor (SM). Ismayilov et al. [32] and Ma et al. [35, 36] have previously created CG solvers in the style of a persistent kernel with GPU-initiated NVSHMEM communication. This is closely related to our work, although there are differences in the underlying SpMV kernels and point-to-point communication schemes. We use a merge-based SpMV kernel to match vendor-optimised kernels from cuSPARSE, whereas Ismayilov et al. [32] uses a naive CSR SpMV kernel and Ma et al. [35, 36] optimises SpMV performance by leveraging tensor cores. Another major difference is the SpMV-related point-to-point communication, where earlier works [32, 35, 36] issue individual get operations (nvshmem_double_g) for each nonzero in the sparse matrix. In this work, we employ a scalable point-to-point communication scheme (see Sections 3.2 and 3.4), avoiding barrier operations during CG iterations and overlapping P2P communication with SpMV. We found these improvements necessary to scale beyond a single node, as earlier works considered only single-node setups with up to 8 GPUs.

6 Conclusion

We have presented several new implementations of CG solvers, including the first multi-GPU implementation that is fully CPU-free and scales beyond a single node. Our benchmarks show excellent performance, both compared to PETSc and to the upper limits derived from the memory bandwidth, although there is room for improvement with regards to the monolithic CG implementation. We anticipate that further optimisation targeting the SpMV component could bring it to the level of the vendor-optimised SpMV kernel provided by cuSPARSE, and thus allow the monolithic CG solver to compete with, or even exceed, the standard offloading approach. Moreover, our results highlight some of the respective strengths and weaknesses of GPU-aware MPI and NCCL/RCCL.

Acknowledgments

This work has received funding through the Inno4scale project, which is funded by the European High-Performance Computing Joint Undertaking (JU) under Grant Agreement No 101118139. The JU receives support from the European Union's Horizon Europe Programme. Authors from Koç University have received funding partly from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

This work is supported by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN)" in Japan (Project ID: jh240030). The work has also benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053. We acknowledge EuroHPC Joint Undertaking for awarding access to the MareNostrum5 supercomputer in Spain and the LUMI supercomputer in Finland (Project IDs: EHPC-DEV-2024D10-091, EHPC-BEN-2025B03-026).

References

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

[2] AMD. 2023. ROCnRDMA. https://github.com/rocmarchive/ROCnRDMA.

[3] AMD. 2023. ROC_SHMEM. https://github.com/ROCm-Developer-Tools/ROC_SHMEM.

[4] AMD. 2025. hipBLAS documentation. https://rocm.docs.amd.com/projects/hipBLAS/en/latest/index.html

[5] AMD. 2025. hipSPARSE User Guide. https://rocm.docs.amd.com/projects/hipSPARSE/en/latest/basics.html

[6] AMD. 2025. RCCL. https://rocm.docs.amd.com/projects/rccl/en/latest/.

[7] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.

[8] Fabio Banchelli, Marta Garcia-Gasulla, Filippo Mantovani, Joan Vinyals, Josep Pocurull, David Vicente, Beatriz Eguzkitza, Flavio CC Galeazzo, Mario C Acosta, and Sergi Girona. 2025. Introducing MareNostrum5: A European pre-exascale energy-efficient system designed to serve a broad spectrum of scientific workloads. arXiv:2503.09917 [cs.DC] https://arxiv.org/abs/2503.09917

[9] Erin Carson and James Demmel. 2014. A residual replacement strategy for improving the maximum attainable accuracy of s-step Krylov subspace methods. *SIAM J. Matrix Anal. Appl.* 35, 1 (2014), 22–43.

[10] Erin Claire Carson. 2015. *Communication-Avoiding Krylov Subspace Methods in Theory and Practice.* Ph. D. Dissertation. University of California, Berkeley.

[11] Erin C. Carson. 2018. The adaptive s-step conjugate gradient method. *SIAM J. Matrix Anal. Appl.* 39, 3 (2018), 1318–1338.

[12] Erin C. Carson, Miroslav Rozložník, Zdeněk Strakoš, Petr Tichý, and Miroslav Tůma. 2018. The numerical stability analysis of pipelined conjugate gradient methods: Historical context and methodology. *SIAM Journal on Scientific Computing* 40, 5 (2018), A3549–A3580.

[13] Anthony T. Chronopoulos and C. William Gear. 1989. On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy. *Parallel computing* 11, 1 (1989), 37–53.

[14] Siegfried Cools, Emrullah Fatih Yetkin, Emmanuel Agullo, Luc Giraud, and Wim Vanroose. 2018. Analyzing the effect of local rounding error propagation on the maximal attainable accuracy of the pipelined conjugate gradient method. *SIAM J. Matrix Anal. Appl.* 39, 1 (2018), 426–450.

[15] NVIDIA Corporation. 2022. *cuSPARSE Library.* NVIDIA Corporation.

[16] NVIDIA Corporation. 2025. *cuBLAS, Release 12.8.* NVIDIA Corporation.

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

[18] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. 2024. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 33, 15 pages. doi:10.1109/SC41406.2024.00039

[19] Jacob Faibussowitsch, Mark F. Adams, Richard Tran Mills, Stefano Zampini, and Junchao Zhang. 2023. Safe, Seamless, And Scalable Integration Of Asynchronous GPU Streams In PETSc. arXiv:2306.17801 [cs.DC] https://arxiv.org/abs/2306.17801

[20] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641.

[21] P. Ghysels and W. Vanroose. 2014. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Comput.* 40, 7 (2014), 224–238. doi:10.1016/j.parco.2013.06.001 7th Workshop on Parallel Matrix Algorithms and Applications.

[22] Taylor Groves, Ben Brock, Yuxin Chen, Khaled Z. Ibrahim, Lenny Oliker, Nicholas J. Wright, Samuel Williams, and Katherine Yelick. 2020. Performance Trade-offs in GPU Communication: A Study of Host and Device-initiated Approaches. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 126–137. doi:10.1109/PMBS51919.2020.00016

[23] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14. doi:10.1109/InPar.2012.6339596

[24] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 609–620. doi:10.1109/SC.2016.51

[25] Khaled Hamidouche and Michael LeBeane. 2020. GPU-initiated OpenSHMEM: correct and efficient intra-kernel networking for dGPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 336–347. doi:10.1145/3332466.3374544

[26] Michael Heroux, Hui Zhou, Ken Raffenetti, Yanfei Guo, Thomas Gillis, Robert Latham, and Rajeev Thakur. 2024. Designing and prototyping extensions to the Message Passing Interface in MPICH. *Int. J. High Perform. Comput. Appl.* 38, 5 (sep 2024), 527–545. doi:10.1177/10943420241263544

[27] M. R. Hestenes and E. Stiefel. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards* 49, 6 (1952), 409–436.

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

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

[30] Intel. 2023. Intel® SHMEM. https://github.com/oneapi-src/ishmem.

[31] Intel. 2025. oneCCL. https://www.intel.com/content/www/us/en/developer/tools/oneapi/oneccl.html.

[32] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 192–202. doi:10.1145/3577193.3593713

[33] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. doi:10.1137/S1064827595287997

[34] Kawthar Shafie Khorassani, Chen-Chun Chen, Hari Subramoni, and Dhabaleswar K. Panda. 2023. Designing and Optimizing GPU-aware Nonblocking MPI Neighborhood Collective Communication for PETSc*. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 646–656. doi:10.1109/IPDPS54959.2023.00070

[35] Tailai Ma, Zhihong Gou, Ningyi Xu, and Shuli Sun. 2024. Efficient Multi-Gpu Implementations of Preconditioned Conjugate Gradient Method Using Tensor Cores. doi:10.2139/ssrn.4985526

[36] Tailai Ma, Zhihong Gou, Ningyi Xu, and Shuli Sun. 2025. Efficient multi-GPU implementations of preconditioned conjugate gradient method. *Advances in Engineering Software* 207 (2025), 103936. doi:10.1016/j.advengsoft.2025.103936

[37] H Martinez-Navarro, B Rodriguez, A Bueno-Orovio, and A Minchole. 2019. Repository for modelling acute myocardial ischemia: simulation scripts and torso-heart mesh. https://ora.ox.ac.uk/objects/uuid:951b086c-c4ba-41ef-b967-c2106d87ee06

[38] Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 678–689. doi:10.1109/SC.2016.57

[39] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Alp Dener, Matthew Knepley, Scott E. Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang. 2021. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Comput.* 108 (2021), 102831. doi:10.1016/j.parco.2021.102831

[40] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Jacob Faibussowitsch, Toby Isaac, Matthew G. Knepley, Todd Munson, Hansol Suh, Stefano Zampini, Hong Zhang, and Junchao Zhang. 2025. PETSc/TAO developments for GPU-based early exascale systems. *The International Journal of High Performance Computing Applications* (2025). doi:10.1177/10943420241303710

[41] Takefumi Miyoshi, Hidetsugu Irie, Keigo Shima, Hiroki Honda, Masaaki Kondo, and Tsutomu Yoshinaga. 2012. FLAT: A GPU Programming Framework to Provide Embedded MPI. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (London, United Kingdom) *(GPGPU-5)*. Association for Computing Machinery, New York, NY, USA, 20–29. doi:10.1145/2159430.2159433

[42] Naveen Namashivayam, Krishna Kandalla, Trey White, Nick Radcliffe, Larry Kaplan, and Mark Pagel. 2022. Exploring GPU Stream-Aware Message Passing using Triggered Operations. arXiv:2208.04817 [cs.DC]

[43] NVIDIA. 2023. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[44] NVIDIA. 2023. NVSHMEM. https://developer.nvidia.com/nvshmem.

[45] NVIDIA. 2025. NCCL. https://developer.nvidia.com/nccl.

[46] Lena Oden, Holger Fröning, and Franz-Joseph Pfreundt. 2014. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 976–983. doi:10.1109/IPDPSW.2014.111

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

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

[49] Sreeram Potluri Pak Markthub, Jim Dinan and Seth Howell. 2022. Improving Network Performance of HPC Systems Using NVIDIA Magnum IO NVSHMEM and GPUDirect Async. https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-and-gpudirect-async/.

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

[51] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, and Dhabaleswar K. Panda. 2012. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 1848–1857. doi:10.1109/IPDPSW.2012.228

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

[53] Didem Unat, Ilyas Turimbetov, Mohammed Kefah Taha Issa, Doğan Sağbili, Flavio Vella, Daniele De Sensi, and Ismayil Ismayilov. 2024. The Landscape of GPU-Centric Communication. arXiv:2409.09874v2 [cs] https://arxiv.org/abs/2409.09874v2

[54] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K. Panda. 2014. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (2014), 2595–2605. doi:10.1109/TPDS.2013.222

[55] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Singh, Sayantan Sur, and D.K. Panda. 2011. MVAPICH2GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development* 26 (06 2011), 257–266. doi:10.1007/s00450-011-0171-3

[56] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. In *2011 IEEE International Conference on Cluster Computing*. 308–316. doi:10.1109/CLUSTER.2011.42

[57] Adam Weingram, Yuke Li, Hao Qi, Darren Ng, Liuyao Dai, and Xiaoyi Lu. 2023. xCCL: A Survey of Industry-Led Collective Communication Libraries for Deep Learning. *Journal of Computer Science and Technology* 38, 1 (Feb 2023), 166–195. doi:10.1007/s11390-023-2894-6

[58] Junchao Zhang, Jed Brown, Satish Balay, Jacob Faibussowitsch, Matthew Knepley, Oana Marin, Richard Tran Mills, Todd Munson, Barry F Smith, and Stefano Zampini. 2021. The PetscSF scalable communication layer. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 842–853.