Concurrent GPU Communications in Krylov Subspace Solvers: Beyond Traditional Overlap

by

Sinan Ekmekçibaşı

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering



KOÇ ÜNİVERSİTESİ

August, 2026

Concurrent GPU Communications in Krylov Subspace Solvers: Beyond Traditional Overlap

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Sinan Ekmekçibaşı

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Didem Unat

_____

Prof. Alptekin Küpçü

_____

Assoc. Prof. Kamer Kaya

Date: _____

ABSTRACT

Concurrent GPU Communications in Krylov Subspace Solvers: Beyond Traditional Overlap

Sinan Ekmekçibaşı

Master of Science in Computer Science and Engineering

August, 2026

Sparse iterative solvers on large GPU clusters are increasingly limited by data movement rather than by arithmetic. A distributed Krylov iteration involves two fundamentally different classes of inter-GPU communication. The first is a latency-bound collective reduction arising from dot products, whose cost grows with the device count. The second is a bandwidth-bound point-to-point (P2P) halo exchange arising from the sparse matrix-vector product. Existing work treats these classes in isolation. Pipelined Krylov methods reformulate the recurrences to hide the global reduction behind computation. A separate line of research overlaps the halo exchange with local work. However, neither approach schedules both communication classes concurrently.

This thesis demonstrates that pipelined solvers offer a richer overlap opportunity than previously exploited. Beyond traditional communication-computation overlap, the pipelined formulation enables communication-communication overlap. We propose a three-stream design for pipelined preconditioned conjugate gradient (Pipe-PCG) and pipelined BiCGSTAB (Pipe-BiCGSTAB). We assign the collective reduction, the halo exchange, and computation to dedicated CUDA streams. We also show that stream separation alone is insufficient because NCCL serializes operations that share a communicator. Achieving genuine concurrency requires more effort than just simply placing computation and communication different streams. Furthermore, we develop an analytical cost model that identifies the balance conditions under which the third stream justifies its additional synchronization.

We evaluate the design on the MareNostrum 5 (H100) and Leonardo (A100) supercomputers, comparing against PETSc and the state-of-the-art two-stream NCCL implementation using SuiteSparse matrices and three massive instances scaled up to 256 GPUs. The three-stream design outperforms the two-stream baseline on every

instance, with the margin widening from 2–6% at 4 GPUs to 18–47% at 64 GPUs. At 64 GPUs, the geometric-mean speedup over the two-stream baseline is 1.33× on MareNostrum 5 and 1.23× on Leonardo for Pipe-PCG, and reaches 1.30× for Pipe-BiCGStab. In large scale experiments, we observed a speedup up to 35% in Pipe-PCG and up to 33% in Pipe-BiCGStab compared to two-stream design. For PETSc, the margin is even wider, up to 3.27x speed-up in both Pipe-PCG and Pipe-BiCGStab.

ÖZETÇE

Krylov Altuzayı Çözücülerinde Eşzamanlı GPU İletişimi: Geleneksel İletişim-İşlem Örtüşmesinin Ötesinde
Sinan Ekmekçibaşı
Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans

Büyük GPU kümelerinde çalışan seyrek iteratif çözücülerin performansı, aritmetik işlemlerden ziyade giderek veri hareketi tarafından sınırlandırılmaktadır. Dağıtık bir Krylov iterasyonu, temelde birbirinden farklı iki tür GPU'lar arası iletişim sınıfını içermektedir. Bunlardan ilki, nokta çarpımlarından kaynaklanan ve maliyeti GPU sayısıyla artan, gecikme ile sınırlanan bir toplu indirgeme işlemidir. İkincisi ise seyrek matris-vektör çarpımından kaynaklanan, bant genişliği ile sınırlanan noktadan noktaya halo değişimidir. Mevcut çalışmalar bu iki iletişim sınıfını birbirinden bağımsız olarak ele almaktadır. Ardışıklaştırılmış Krylov yöntemleri, yineleme bağıntılarını yeniden düzenleyerek küresel indirgeme işlemini hesaplamanın arkasına gizlemektedir. Ayrı bir araştırma yaklaşımı ise halo değişimini yerel hesaplamalarla örtüştürmektedir. Ancak bu yaklaşımların hiçbiri her iki iletişim sınıfını eş zamanlı olarak planlamamaktadır.

Bu tez, ardışıklaştırılmış çözücülerin bugüne kadar yararlanılandan daha geniş bir örtüşme olanağı sunduğunu göstermektedir. Geleneksel iletişim-hesaplama örtüşmesinin ötesinde, ardışıklaştırılmış formülasyon iletişim-iletişim örtüşmesini de mümkün kılmaktadır. Bu kapsamda, ardışıklaştırılmış önkoşullandırılmış eşlenik gradyan (Pipe-PCG) ve ardışıklaştırılmış BiCGSTAB (Pipe-BiCGSTAB) yöntemleri için üç akışlı (three-stream) bir tasarım önerilmektedir. Toplu indirgeme, halo değişimi ve hesaplama işlemleri ayrı CUDA akışlarına atanmıştır. Ayrıca, yalnızca akışların ayrılmasının yeterli olmadığı; NCCL'nin aynı iletişimciyi (NCCL communicator) paylaşan işlemleri ardışık olarak yürüttüğü gösterilmiştir. Gerçek anlamda eşzamanlılık elde etmek, hesaplama ve iletişim işlemlerini farklı akışlara yerleştirmenin ötesinde ek düzenlemeler gerektirmektedir. Bunun yanı sıra, üçüncü akışın getirdiği ek senkronizasyon maliyetini hangi koşullar altında telafi ettiğini belirleyen analitik bir maliyet modeli de geliştirilmiştir.

Önerilen tasarımı, 256 GPU'ya kadar ölçeklenen SuiteSparse matrisleri ve üç

büyük ölçekli problem örneği kullanarak MareNostrum 5 (H100) ve Leonardo (A100) süperbilgisayarlarında değerlendiriyoruz. Performansımızı PETSc ve mevcut en gelişmiş iki akışlı NCCL uygulamasıyla karşılaştırıyoruz. Üç akışlı tasarım, her problem örneğinde iki akışlı temel yaklaşımı geride bırakmakta ve performans farkı 4 GPU'da %2–6 seviyesinden 64 GPU'da %18–47 seviyesine kadar genişlemektedir. 64 GPU'da, iki akışlı temel yaklaşıma göre geometrik ortalama hızlanma Pipe-PCG için MareNostrum 5 üzerinde 1,33× ve Leonardo üzerinde 1,23× olarak ölçülürken, Pipe-BiCGSTAB için 1,30× seviyesine ulaşmaktadır. Büyük ölçekli deneylerde, iki akışlı tasarıma kıyasla Pipe-PCG'de %35'e ve Pipe-BiCGSTAB'de %33'e varan hızlanmalar gözlemledik. PETSc ile karşılaştırıldığında ise performans farkı daha da genişlemekte ve hem Pipe-PCG hem de Pipe-BiCGSTAB için 3,27 kata kadar hızlanma elde edilmektedir.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Assoc. Prof. Didem Unat, for her invaluable guidance, unwavering support, and encouragement throughout my master's studies and the completion of this thesis. Her expertise and exceptional mentorship have played a pivotal role in my development as a researcher, beginning from my very first days as an undergraduate intern under her supervision.

I would also like to thank the members of ParCoreLab for their support, collaboration, and for fostering a stimulating research environment. I am grateful to Simula Research Laboratory for hosting me during my research visit and to our collaborators, Xing Cai, Johannes Langguth, and James Trotter, for their valuable guidance and contributions to this work. Finally, I would like to thank everyone who has contributed, directly or indirectly, to my academic journey and the completion of this thesis.

Finally, I would like to express my deepest gratitude to my family and friends for their unwavering support, patience, and encouragement throughout this journey. Their belief in me has been an invaluable source of motivation.

TABLE OF CONTENTS

List of Tables **xi**

List of Figures **xii**

Abbreviations **xiv**

Chapter 1: Introduction **1**

1.1 Communication in Krylov Solvers 2

1.2 The Gap and Thesis Statement 3

1.3 Contributions 4

1.4 Thesis Outline 5

Chapter 2: Background and Motivation **7**

2.1 Krylov Subspace Solvers and the Conjugate Gradient Method 7

2.2 Distributed-Memory Parallelization 9

2.3 GPU Communication Mechanisms 11

2.4 Pipelined Preconditioned Conjugate Gradient 12

2.5 Pipelined BICGSTAB 14

2.6 Motivation: Beyond Traditional Overlap 16

Chapter 3: Multi-GPU Implementation of Krylov Subspace Solvers **21**

3.1 Multi-GPU Pipelined PCG and Pipelined BICGSTAB 22

3.1.1 Computation 22

3.1.2 Communication 26

3.2 Two-Stream Baseline 28

3.3 Three-Stream Approach 31

3.4 Alternative Communication Libraries 33

Chapter 4: Analytical Cost Model and Performance Implications 35

4.1 Notation . . . 35

4.2 Analytical Cost Model for Pipelined PCG . . . 36

4.2.1 Cost Model of Two-Stream Execution . . . 36

4.2.2 Cost Model of Three-Stream Execution . . . 37

4.2.3 Performance Implications . . . 39

4.3 Analytical Cost Model for Pipelined BiCGStab . . . 41

4.3.1 Structural Differences and Notation . . . 41

4.3.2 Communication Serialization and the Halo Overlap Window . 42

4.3.3 Cost Model of Two-Stream Execution . . . 43

4.3.4 Cost Model of Three-Stream Execution . . . 44

4.3.5 Gain of the Third Stream . . . 44

4.3.6 Performance Implications . . . 45

4.4 Resource Contention . . . 46

Chapter 5: Experiments and Evaluation 48

5.1 Experimental Setup . . . 48

5.1.1 Supercomputers . . . 49

5.1.2 Software Stack . . . 50

5.1.3 Test Matrices . . . 50

5.1.4 Measurement Methodology . . . 51

5.2 Results . . . 52

5.3 Performance Analysis . . . 61

5.3.1 How we measure . . . 61

5.3.2 Where the compute time goes . . . 63

5.3.3 Why the third stream pays off at scale . . . 64

5.3.4 From communication savings to end-to-end speedup . . . 65

Chapter 6: Related Work 67

6.1 Pipelined and Communication-Avoiding Krylov Methods . . . 67

6.2 Halo and P2P Overlap in Iterative Solvers and Multigrid 68
6.3 GPU-Side Communication Strategies for CG 68
6.4 Joint Scheduling of Communication Classes 69

Chapter 7: Conclusion and Future Work **70**
7.1 Conclusion 70
7.2 Limitations 71
7.3 Future Work 71

Bibliography **73**

LIST OF TABLES

2.1 Latency (in microseconds) of a `ncclAllReduce` operation reported by the `nccl-tests` benchmarks on MareNostrum 5 for a 16-byte message (two double-precision values) using in-place reduction. 17

2.2 Throughput (in MB/s) and latency (in microsecs) of internode point-to-point communication with `ncclSend`/`ncclRecv` with different message sizes, as reported by the `nccl-tests` benchmarks on 2 H100 GPUs on MareNostrum5. (Theoretical max bandwidth is 25 GB/s.) . 17

2.3 Throughput (in GB/s) as reported by the `BabelStream` benchmark on a MareNostrum 5 ACC node with an NVIDIA H100 64 GB GPU that has a theoretical peak memory bandwidth of 1,631.2 GB/s with active ECC. 18

5.1 Sparse matrices used in experiments. 51

5.2 Speed-ups of three-stream (ours) over PETSc and two-stream on MareNostrum5 (H100) and Leonardo (A100) using 64 GPUs in Pipe-PCG. 56

5.3 Speed-ups of three-stream (ours) over PETSc and two-stream for Pipe-BiCGStab on MareNostrum5 (H100) at 16, 32, and 64 GPUs. . 60

5.4 Kernel breakdown for Queen_4147 on MareNostrum 5. 62

5.5 Reduction cost against overlappable work for Queen_4147. 64

LIST OF FIGURES

2.1 NVIDIA Nsight Systems traces for the three-stream microbenchmark on 4 GPUs. **(a)** Two-stream baseline, in which the allreduce and DAXPY operations share the Compute stream alongside a separate P2P stream. **(b)** Proposed three-stream approach, showing clear concurrent overlap across dedicated Compute, P2P, and Allreduce streams. 19

3.1 Two-stream versus three-stream design of pipelined preconditioned CG. The two-stream design overlaps the P2P halo exchange with computation, while the three-stream design additionally offloads the collective `allreduce` onto a dedicated stream so that it overlaps with both computation and P2P communication. 28

3.2 Two-stream versus three-stream design of pipelined BICGSAB. The two-stream design overlaps the P2P halo exchange with computation, while the three-stream design additionally offloads the collective `allreduce` onto a dedicated stream so that it overlaps with both computation and P2P communication. 29

3.3 NVIDIA Nsight Systems profiling of the naive three-stream design versus our three-stream design. In (a), a single NCCL communicator serializes the collective and P2P operations, so no real overlap occurs; in (b), splitting the communicator with `ncclCommSplit` lets the collective, P2P, and compute streams run concurrently. 33

5.1 Three-stream vs. PETSc and two-stream on SuiteSparse matrices on MareNostrum5. 53

5.2 Three-stream vs. PETSc and two-stream on SuiteSparse matrices on Leonardo. 54

5.3 Three-stream vs. PETSc and two-stream on SuiteSparse matrices on MareNostrum5. 55
5.4 Three-stream vs. PETSc and two-stream on the larger matrices on MareNostrum5. 57
5.5 Three-stream vs. PETSc and two-stream on the larger matrices on Leonardo. 58
5.6 Three-stream vs. PETSc and two-stream on the larger matrices on MareNostrum5. 58

ABBREVIATIONS

API	Application Programming Interface
AXPY	Scalar Alpha X Plus Y (vector update)
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUSPARSE	CUDA Sparse Matrix Library
CG	Conjugate Gradient
CPU	Central Processing Unit
CTA	Cooperative Thread Array
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HPC	High Performance Computing
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communication Library
NIC	Network Interface Card
NVSHMEM	NVIDIA Symmetric Hierarchical Memory
P2P	Point-to-Point
PCG	Preconditioned Conjugate Gradient
Pipe-PCG	Pipelined Preconditioned Conjugate Gradient
Pipe-BICGSTAB	Pipelined Stabilized BiConjugate Gradient
RCCL	ROCm Collective Communication Library
RDMA	Remote Direct Memory Access
RMA	Remote Memory Access
SM	Streaming Multiprocessor
SpMV	Sparse Matrix-Vector Multiplication

Chapter 1

INTRODUCTION

Large-scale parallel execution on supercomputers is essential for computational science and machine learning. Modern high-performance computing (HPC) systems achieve their peak performance using dense GPU nodes, and today's largest machines connect thousands of these accelerators through high-bandwidth networks. Because GPUs offer immense arithmetic throughput, the main performance bottleneck for scientific applications has shifted from raw computation to data movement between devices. When we decompose problems across many GPUs, the unavoidable inter-process communication used to coordinate tasks and exchange data often becomes the dominant cost. This communication overhead limits how far a parallel application can scale. We reach this ceiling sooner on GPU systems because each device finishes its computation so quickly that even modest communication delays create large relative overheads. Therefore, mitigating this communication bottleneck is essential for sustaining performance at scale.

Overlapping communication with computation is a widely used approach to reduce this cost. For example, applications commonly use non-blocking send and receive operations to transfer messages, typically between neighboring processes exchanging halo data [Spies et al., 2022, Horikoshi et al., 2022]. When these messages progress in the background, useful computation proceeds concurrently. This allows the communication time to be at least partially hidden behind the computation time. This strategy works well in both multi-CPU and multi-GPU architectures. On CPU systems, messages typically progress through dedicated communication threads [Langguth et al., 2015] or advanced MPI implementations that support asynchronous progress [Ruhela et al., 2018]. On GPU systems, we usually achieve

this overlap using concurrent streams for communication and computation [Trotter et al., 2025, Sağbili et al., 2025]. Placing communication and computation on separate streams allows the hardware to execute them simultaneously, as long as the operations are independent and the communication library supports asynchronous progress.

1.1 Communication in Krylov Solvers

The conjugate gradient (CG) method [Hestenes and Stiefel, 1952] and its preconditioned variants (PCG) are canonical Krylov subspace solvers for symmetric positive definite sparse linear systems. These systems arise frequently when discretizing partial differential equations and in many other numerical models. Therefore, CG and PCG serve as central building blocks in large-scale scientific applications. Their scalability directly determines how far we can push many simulations. A distributed CG iteration relies on two main computational kernels: sparse matrix-vector multiplication (SpMV) and vector operations like dot products and vector updates. Each kernel creates a distinct communication pattern.

The first communication class is global collective communication. This arises from the dot products used to compute the scalar coefficients. To evaluate a dot product over a distributed vector, every process must contribute a partial sum and receive the global result. We realize this operation as an `allreduce` collective across all participating GPUs. This communication involves all processes, carries a small volume of data, and is strictly latency-bound. Its cost grows with the number of GPUs, which makes it increasingly expensive as the system scales.

The second communication class is point-to-point (P2P) halo exchange, which comes from the distributed SpMV. When we partition the sparse matrix across GPUs, each device must fetch remote vector entries from its neighbors to resolve off-diagonal nonzeros. This exchange is irregular, involves only a subset of processes, and moves a relatively large volume of data. Therefore, it is bandwidth-bound rather than latency-bound. These two classes sit at opposite ends of the communication spectrum: low-volume, latency-bound collectives versus high-volume, bandwidth-

bound neighbor exchanges.

Prior work on communication overlap in CG generally treats these two communication types separately. The literature on pipelined Krylov methods targets the collective operations. These works seek to lower global synchronization overhead by reformulating the algorithm. This reformulation allows us to overlap a non-blocking reduction with the SpMV and preconditioner operations. Researchers extend this idea using deeper pipelines or variants that require fewer reductions per iteration [Ghysels and Vanroose, 2014, Cornelis et al., 2018, Tiwari and Vadhiyar, 2020, Sanan et al., 2016].

In principle, the pipelined CG algorithm introduced by Ghysels and Vanroose [Ghysels and Vanroose, 2014] can overlap both collective and P2P communication with computation. In practice, however, that work and its successors focused only on hiding the collective reductions. They did not exploit the concurrent P2P opportunity. A separate line of research focuses on overlapping the halo/P2P data exchange with computation in iterative solvers and multigrid methods. These efforts primarily rely on scheduling and implementation techniques, such as specialized OpenMP threads [Nakajima et al., 2022, Nakajima, 2025]. On GPUs, more recent studies improve the communication path using multi-GPU SpMV overlap, persistent kernels, and CPU- or GPU-initiated communication mechanisms [Bernaschi et al., 2025, Ismayilov et al., 2023, Trotter et al., 2025, Ma et al., 2025].

1.2 The Gap and Thesis Statement

Despite this substantial body of work, none of these approaches jointly schedule collective communication, halo exchange, and computation in a single concurrent design. Each existing technique overlaps only one class of communication with computation, leaving the other serialized. This represents a missed opportunity. The two communication classes are independent and have complementary bottlenecks. One is latency-bound, while the other is bandwidth-bound. Therefore, running them concurrently avoids the resource contention that occurs when overlapping two similar operations.

This thesis shows that pipelined Krylov subspace solvers offer a richer overlap opportunity than previously exploited. Beyond traditional communication-computation overlap, the pipelined formulation enables communication-communication overlap. In this approach, collective reductions and P2P halo exchanges proceed concurrently alongside ongoing computation. This opportunity arises naturally in pipelined iterative solvers, which are essential for solving large-scale sparse linear systems. To exploit this, we propose a novel three-stream implementation of pipelined preconditioned CG and pipelined stabilized biconjugate gradient methods. Our design treats collective reduction, halo exchange, and SpMV as three distinct, concurrently scheduled components. We implement this design using the NVIDIA Collective Communication Library (NCCL) [NVIDIA Corporation, 2026b], which provides native stream support for GPU communication. As we discuss in later chapters, simply placing operations on separate streams does not guarantee overlap. Achieving genuine concurrency requires careful configuration and the use of separate communicators. This ensures the two communication streams make independent progress without blocking each other.

1.3 Contributions

The contributions of this thesis are as follows.

- We propose a three-stream overlap strategy for pipelined preconditioned CG and pipelined stabilized biconjugate gradient that concurrently overlaps collective communication, P2P communication, and computation, extending conventional computation-communication overlap into computation-communication-communication overlap across multiple GPUs.

- We develop an analytical cost model that compares the two-stream (traditional) design with the proposed three-stream design, characterizes the regimes in which the three-stream approach is beneficial, and guides the design of our experiments.

- We evaluate the proposed three-stream strategy on a range of sparse matrices and compare it against both PETSc and the state-of-the-art two-stream NCCL implementation of Trotter et al. [Trotter et al., 2025]. The three-stream design achieves up to 35% speedup on 256 GPUs on MareNostrum 5 and up to 47% speedup on 64 GPUs on Leonardo over the state-of-the-art two-stream method [Trotter et al., 2025].

Although this thesis demonstrates the three-stream design for pipelined Krylov subspace solvers, the underlying concept is much more general. Any application combining high-bandwidth local data exchange with high-latency global coordination can benefit from this triple overlap. For example, these patterns appear in optimization and simulation workloads that interleave neighbor exchanges with periodic convergence checks or global best-solution updates. Therefore, the three-stream principle serves as a template for a broad class of distributed applications, rather than a technique limited to a single type of solvers.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the background and motivation, reviewing the conjugate gradient and pipelined CG methods, the two communication classes that arise in a distributed solve, and the GPU communication mechanisms and libraries on which our work builds; it also motivates communication-communication overlap through microbenchmarks and profiling.

Chapter 3 presents our multi-GPU implementation of pipelined preconditioned CG, describing how the solver is mapped onto three streams and how NCCL communicators and configuration options are used to allow the collective and P2P streams to make independent progress. Chapter 4 introduces the analytical cost model for the two-stream and three-stream executions and analyzes the performance implications of the model across different parameter regimes.

Chapter 5 reports our experimental methodology and results on the MareNostrum 5 and Leonardo supercomputers, comparing the three-stream design against

the two-stream state of the art and PETSc across a range of sparse matrices and GPU counts, and analyzing the observed scaling behavior.

Chapter 6 surveys related work on pipelined Krylov methods, communication overlap, and GPU communication, and locates our contribution within it.

Finally, Chapter 7 summarizes the thesis, discusses its limitations, and outlines directions for future work.

Chapter 2

BACKGROUND AND MOTIVATION

In this chapter, we introduce the background needed to understand the remainder of the thesis and motivate the work. We begin with the conjugate gradient (CG) method and its place among Krylov subspace solvers, focusing on the two kernels that dominate its communication behavior, dot products and sparse matrix–vector multiplication. We then describe how these kernels are parallelized across a distributed-memory multi-GPU system, and which communication mechanisms modern GPU clusters expose to implement them. Next, we present the pipelined preconditioned CG (Pipe-PCG) and pipelined stabilized biconjugate gradient methods, which reorganize the classical algorithm to relax its synchronization bottleneck. Finally, we motivate our contribution with a set of microbenchmarks that quantify the relevant computation and communication costs and demonstrate that overlapping collective communication, point-to-point (P2P) communication, and computation concurrently, rather than in pairs, results with measurable performance gains.

2.1 Krylov Subspace Solvers and the Conjugate Gradient Method

A large fraction of scientific and engineering applications ultimately reduces to solving a linear system

$$Ax = b, \tag{2.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a large, sparse coefficient matrix, $b \in \mathbb{R}^n$ is a known right-hand side, and $x \in \mathbb{R}^n$ is the unknown solution vector.

When the matrix A is symmetric positive definite (SPD), the conjugate gradient (CG) method [Hestenes and Stiefel, 1952] is our solver of choice. We can view CG as minimizing the energy norm of the error over the Krylov subspace. It features a short-recurrence formulation. Each iteration updates the approximate solution x_i,

the residual r_i, and a search direction p_i using only a few vector operations and a single application of A. Because these recurrences are short, CG only stores a small, fixed number of vectors, regardless of the iteration count. This low memory footprint makes it highly popular on memory-constrained accelerators.

In exact arithmetic, CG converges in at most n iterations. However, its practical convergence rate depends heavily on the spectral properties of A. An ill-conditioned matrix—one with a large ratio between its largest and smallest eigenvalues—causes CG to converge slowly. To accelerate convergence, we apply a preconditioner M. This matrix approximates A but remains cheap to invert. The preconditioned conjugate gradient (PCG) method implicitly solves the equivalent system $M^{-1}Ax = M^{-1}b$. This allows CG to operate on a better-conditioned system, which reduces the iteration count significantly. The trade-off is one additional kernel per iteration, which is the application of M^{-1}. A simple, widely used option is the Jacobi (diagonal) preconditioner, $M = \text{diag}(A)$. We compute its inverse by taking the reciprocals of the diagonal entries, which adds minimal computational overhead while improving the conditioning for many problems.

For this thesis, the crucial observation is that we build each CG or PCG iteration from a small set of computational kernels. These include vector updates of the form $y \leftarrow \alpha x + y$ (AXPY), inner products (x, y) (DOT), the application of the preconditioner, and the sparse matrix-vector multiplication (SpMV) $y = Ax$. The AXPY kernels and the preconditioner are embarrassingly parallel and require no inter-process communication. In contrast, two kernels, the SpMV and the DOT products, form the communication behavior of the solver.

While CG and PCG dominate symmetric problems, the Biconjugate Gradient Stabilized (BiCGSTAB) method is widely preferred for non-symmetric matrices. It provides smoother convergence than earlier non-symmetric solvers and avoids the computationally expensive transpose matrix-vector product $A^T x$. However, network latency constrains a standard BiCGSTAB iteration even more tightly than CG. Each iteration requires two SpMVs and multiple independent global synchronization points.

To address these non-symmetric systems, pipelined BiCGSTAB (Pipe-BiCGSTAB)

extends the latency-hiding paradigm of pipelined CG (Pipe-CG). By fundamentally rearranging the mathematical recurrence relations, Pipe-BiCGSTAB groups the inner products and delays their application. This restructuring breaks immediate data dependencies. Consequently, we can asynchronously overlap the global reductions with the dual SpMVs and local vector operations (like AXPY updates). This effectively hides the communication overhead behind memory-bound computations, which allows the solver to achieve superior parallel efficiency and strong scaling on modern high-performance computing architectures.

Basically, two kernels form the communication behavior of the solver once it is distributed across many devices:

- **Dot products (DOT).** An inner product reduces two vectors to a single scalar, whose value is needed to compute the scalar coefficients that steer the next iteration. Because every process holds only a slice of the vectors, computing a global inner product requires summing partial results across all processes, i.e., a *global reduction*.

- **Sparse matrix–vector multiplication (SpMV).** Multiplying the distributed matrix A by a distributed vector requires each process to obtain the vector entries that couple to its local rows but are owned by other processes, i.e., an irregular exchange of *halo* data.

The remainder of this chapter examines how these two kernels shape the design of a scalable multi-GPU solver.

2.2 Distributed-Memory Parallelization

To solve a system too large for a single GPU, we must partition both the matrix and the associated vectors across devices. We adopt a row-wise partitioning of the sparse matrix A. We assign each process (mapped to one GPU) a contiguous block of matrix rows and the corresponding segments of all iteration vectors. Once this distribution is fixed, the AXPY kernels, the preconditioner application, and the local inner products operate entirely on data resident in a GPU's own memory. However,

the two communication-related kernels mentioned in Section 2.1 require processes to exchange data.

Dot products and collective reductions. Under row-wise partitioning, we compute a global inner product (x, y) in two stages. First, each process computes the partial inner product of its local vector segments. Next, these partial results reduced into a single global value available to every process. This is exactly an allreduce collective operation over a very small payload—typically two or four double-precision scalars per dot product, depending on the solver type. Although the message is tiny, the operation exposes a global synchronization point. Every process must contribute its partial sum and wait for the combined result before proceeding to the next computations. As the number of participating GPUs increases, the latency of this collective grows.

SpMV and irregular halo exchange. The second communication pattern arises from the distributed SpMV $y = Ax$. Each GPU owns only a slice of x. Because its assigned rows typically have nonzero columns referencing entries of x owned by other GPUs, computing the local part of y requires fetching these remote entries. The matrix's sparsity structure determines this required set of remote entries. Consequently, this exchange is irregular, a process communicates only with the neighbors that own its referenced columns, and message volumes vary between neighbors. To structure this computation, we split each process's local matrix partition into two blocks. The diagonal block couples only to locally owned vector entries, while the off-diagonal block couples to remote entries. A process can multiply the diagonal block with its local data while simultaneously executing the point-to-point transfers to fetch remote entries. Once the halo data arrives, the process multiplies the off-diagonal block. This split exposes an overlap opportunity between local computation and halo communication, which we exploit throughout this work. We discuss the mechanics of packing, transferring, and unpacking halo data, along with the stream and event structure that realizes this overlap, in Chapter 3.

Together, these two patterns give distributed CG a distinctive communication

profile, a latency-bound global collective for every dot product, and a bandwidth-bound irregular P2P exchange for every SpMV. The interaction between these two fundamentally different communication types forms the central contribution of this thesis.

2.3 GPU Communication Mechanisms

Realizing the communication patterns discussed above efficiently on multi-GPU settings requires an execution model that keeps the GPUs busy and avoids unnecessary intervention of the host CPU. Modern GPUs execute work asynchronously with respect to the CPU through *streams*, a CUDA stream is an ordered queue of operations that execute in issue order relative to one another, while operations in different streams may execute concurrently, depending on hardware resource availability. This makes streams the natural choice for overlapping independent work, since kernels and communication routines placed on distinct streams can run at the same time, concurrently. Dependencies between streams are expressed with *CUDA events*, a stream can record an event at a point in its queue, and another stream can be made to wait on that event before continuing. Using `cudaEventRecord` and `cudaStreamWaitEvent` to enforce cross-stream ordering keeps the coordination entirely on the device and avoids the host-side synchronization.

For inter-GPU communication, we build on the NVIDIA Collective Communication Library (NCCL) [NVIDIA Corporation, 2026b]. NCCL provides topology-aware, highly optimized implementations of collective operations such as allreduce, as well as point-to-point `ncclSend` and `ncclRecv` primitives. Its most important property for our purposes is that it is *stream-aware*, which means communication routines are enqueued on a CUDA stream just like ordinary kernels, so they participate in the same asynchronous execution model and can be overlapped with computation and with one another using events. This lets an application queue packing, sending, receiving, and unpacking operations from the host without incurring CPU–GPU synchronization overhead. Prior work by Trotter et al. [Trotter et al., 2025] and by De Sensi et al. [De Sensi et al., 2024] has shown that on modern

supercomputers NCCL delivers performance comparable to GPU-aware MPI, which makes it an appealing choice since it also has stream property as mentioned above.

We briefly highlight two alternative communication libraries here and examine them in detail in Chapter 3. GPU-aware MPI can, in principle, express the same patterns through non-blocking collectives (`MPI_Iallreduce`) and non-blocking point-to-point calls (`MPI_Isend`/`MPI_Irecv`) operating directly on GPU buffers. However, support for non-blocking reductions on GPU buffers is limited in some MPI implementations, including OpenMPI [Community, 2026], which can force data to be staged through host buffers. NVSHMEM, in contrast, follows a one-sided partitioned global address space model in which a process directly reads from or writes to remote GPU memory using put/get and signalling primitives, several of which are stream-aware. Expressing an irregular halo exchange in this model is possible but requires careful use of remote memory access (RMA) and signalling operations, as discussed by Trotter et al. [Trotter et al., 2025] and in the Uniconn communication library [Sağbili et al., 2025].

2.4 Pipelined Preconditioned Conjugate Gradient

Standard PCG performs two separate dot products per iteration, and in a distributed-memory setting each of them forces a global synchronization. These frequent synchronizations expose the full network latency of the allreduce collective and cause some processes to sit idle while they wait for the reduced value, producing a start-and-stop execution pattern that severely inhibits parallel efficiency at large GPU counts [Ghysels and Vanroose, 2014]. Preconditioning reduces the number of iterations but does nothing to relieve this per-iteration synchronization cost; on the contrary, it inserts an additional kernel that must be scheduled around the reductions.

The pipelined preconditioned conjugate gradient (Pipe-PCG) method, introduced by Ghysels and Vanroose [Ghysels and Vanroose, 2014], directly targets this synchronization bottleneck. By introducing a set of auxiliary vectors and mathematically restructuring the recurrences, Pipe-PCG groups the two dot products

of an iteration into a single reduction, thus reducing global synchronization to one non-blocking phase per iteration. More importantly, this reformulation rearranges the computation to postpone the synchronization for this reduction. We initiate the global sum and overlap it with independent work in the same iteration, namely, the preconditioner application and the SpMV. Instead of blocking until the scalars are ready, we hide the latency of the collective behind concurrent computation, removing it from the critical path.

Algorithm 1 Pipelined Preconditioned CG (Pipe-PCG).

1: $r_0 := b - Ax_0$; $u_0 := M^{-1}r_0$; $w_0 := Au_0$
2: **for** $i = 0, \ldots$ **do**
3: $\quad \gamma_i := (r_i, u_i)$
4: $\quad \delta := (w_i, u_i)$
5: $\quad m_i := M^{-1}w_i$
6: $\quad n_i := Am_i$
7: $\quad$ **if** $i > 0$ **then**
8: $\quad\quad \beta_i := \gamma_i/\gamma_{i-1}; \quad \alpha_i := \gamma_i/(\delta - \beta_i\gamma_i/\alpha_{i-1})$
9: $\quad$ **else**
10: $\quad\quad \beta_i := 0; \quad \alpha_i := \gamma_i/\delta$
11: $\quad$ **end if**
12: $\quad z_i := n_i + \beta_i z_{i-1}$
13: $\quad q_i := m_i + \beta_i q_{i-1}$
14: $\quad s_i := w_i + \beta_i s_{i-1}$
15: $\quad p_i := u_i + \beta_i p_{i-1}$
16: $\quad x_{i+1} := x_i + \alpha_i p_i$
17: $\quad r_{i+1} := r_i - \alpha_i s_i$
18: $\quad u_{i+1} := u_i - \alpha_i q_i$
19: $\quad w_{i+1} := w_i - \alpha_i z_i$
20: **end for**

Algorithm 1 lists the resulting iteration. Pipe-PCG relies on the same funda-

mental compute kernels as standard PCG, namely AXPY, DOT, SpMV, and the preconditioner, but the restructuring introduces additional vector updates. Concretely, a single Pipe-PCG iteration performs the two dot products γ_i and δ as one fused reduction (lines 3–4), eight AXPY operations to update the vectors z_i, q_i, s_i, p_i, x_{i+1}, r_{i+1}, u_{i+1}, and w_{i+1} (lines 11–18), one SpMV (line 6), and a single preconditioner application (line 5). The main computational difference from standard PCG is the use of eight AXPYs instead of three. Although these extra vector updates increase the per-iteration computational load, the benefit of reducing global synchronization to a single, overlappable phase amortizes this cost during large-scale executions [Ghysels and Vanroose, 2014].

2.5 Pipelined BICGSTAB

Standard BiCGStab performs four dot products per iteration, distributed over three separate global reduction phases that produce the scalars α_i, ω_i and β_i. In a distributed-memory setting each of these phases forces a global synchronization, so a single BiCGStab iteration exposes the network latency of the allreduce collective three times rather than twice as in CG. The Furthermore, the reduction producing ω_i separates the two sparse matrix-vector products in each iteration. Because both SpMVs depend on a preceding synchronization, we cannot use either one to hide communication time. This causes some processes to sit idle while they wait for the reduced values, producing a start-and-stop execution pattern that severely inhibits parallel efficiency at large GPU counts [Cools and Vanroose, 2017].

The pipelined BiCGStab (Pipe-BiCGStab) method, introduced by Cools and Vanroose [Cools and Vanroose, 2017], directly targets this synchronization bottleneck. By introducing a set of auxiliary vectors and mathematically restructuring the recurrences. Pipe-BiCGStab first makes the scalar α_i independent of the intermediate vectors of its own iteration, which allows its dot product to be merged with the reduction that computes β_i and lowers the number of global synchronizations from three to two. More importantly, this reformulation restructures the computation to decouple the reductions from immediate synchronization. We launch each global

sum as a non-blocking collective and overlap it with the subsequent SpMV, avoiding a blocking wait for the scalars. In other words, the latency of both collectives can be hidden behind computation instead of being exposed on the critical path.

Algorithm 2 Pipelined BiCGStab (Pipe-BiCGStab).

1: $r_0 := b - Ax_0$; $w_0 := Ar_0$; $t_0 := Aw_0$
2: $\alpha_0 := (r_0, r_0)/(r_0, w_0)$; $\beta_{-1} := 0$
3: **for** $i = 0, \ldots$ **do**
4: $\quad p_i := r_i + \beta_{i-1}(p_{i-1} - \omega_{i-1}s_{i-1})$
5: $\quad s_i := w_i + \beta_{i-1}(s_{i-1} - \omega_{i-1}z_{i-1})$
6: $\quad z_i := t_i + \beta_{i-1}(z_{i-1} - \omega_{i-1}v_{i-1})$
7: $\quad q_i := r_i - \alpha_i s_i$
8: $\quad y_i := w_i - \alpha_i z_i$
9: $\quad$ **begin reduction** (q_i, y_i); (y_i, y_i)
10: $\quad\quad v_i := Az_i$
11: $\quad$ **end reduction**
12: $\quad \omega_i := (q_i, y_i)/(y_i, y_i)$
13: $\quad x_{i+1} := x_i + \alpha_i p_i + \omega_i q_i$
14: $\quad r_{i+1} := q_i - \omega_i y_i$
15: $\quad w_{i+1} := y_i - \omega_i(t_i - \alpha_i v_i)$
16: $\quad$ **begin reduction** (r_0, r_{i+1}); (r_0, w_{i+1}); (r_0, s_i); (r_0, z_i)
17: $\quad\quad t_{i+1} := Aw_{i+1}$
18: $\quad$ **end reduction**
19: $\quad \beta_i := (\alpha_i/\omega_i)(r_0, r_{i+1})/(r_0, r_i)$
20: $\quad \alpha_{i+1} := (r_0, r_{i+1})/\big((r_0, w_{i+1}) + \beta_i(r_0, s_i) - \beta_i\omega_i(r_0, z_i)\big)$
21: **end for**

Algorithm 2 lists the resulting iteration, in which r_0 denotes the fixed shadow residual. Pipe-BiCGStab relies on the same fundamental compute kernels as standard BiCGStab, namely AXPY, DOT, and SpMV, but the restructuring introduces additional vector updates. Concretely, a single Pipe-BiCGStab iteration performs

six dot products grouped into two fused reductions (lines 9 and 16), eight AXPY operations to update the vectors p_i, s_i, z_i, q_i, y_i, x_{i+1}, r_{i+1} and w_{i+1} (lines 4–8 and 13–15), and two SpMVs (lines 10 and 17). The main computational difference from standard BiCGStab is the use of eight AXPYs instead of four, together with a larger set of auxiliary vectors that must be kept resident, which raises the vector storage requirement from seven to eleven. Although these extra vector updates increase the per-iteration computational load, the benefit of reducing global synchronization to two overlappable phases amortizes this cost during large-scale executions [Cools and Vanroose, 2017]. The additional recurrences also make the method more sensitive to the propagation of local rounding errors, which lowers the maximal attainable accuracy. This is commonly addressed by periodically resetting the residual and auxiliary vectors to their true values through a residual replacement strategy [Cools and Vanroose, 2017, Cools, 2019].

2.6 Motivation: Beyond Traditional Overlap

The conventional strategy for hiding communication latency is to overlap computation with a single communication operation at a time. Pipe-PCG, however, issues *two* distinct communication types per iteration, a global allreduce for the dot products and an irregular P2P halo exchange for the SpMV, alongside its compute kernels. This raises a natural question: rather than overlapping computation with only one communication operation, can we overlap collective communication, P2P communication, and computation *concurrently* to achieve further performance gains beyond traditional latency hiding?

To answer this, we first need a sense of the relative costs of the operations involved. Tables 2.1–2.3 report a set of microbenchmark measurements collected on the MareNostrum 5 supercomputer that characterize the cost of the collective, the P2P transfer, and the compute kernels of a Pipe-PCG iteration.

Table 2.1 shows that on 64 GPUs an allreduce over two double-precision values, which is exactly the collective needed to complete the fused dot product of a Pipe-PCG iteration, takes about $45.4\,\mu s$. This is the latency that the solver must hide to

Table 2.1: Latency (in microseconds) of a `ncclAllReduce` operation reported by the `nccl-tests` benchmarks on MareNostrum 5 for a 16-byte message (two double-precision values) using in-place reduction.

Nodes	No. of GPUs	Size [B]	Latency [µs]
1	4	16	8.32
2	8	16	16.36
4	16	16	24.82
8	32	16	33.01
16	64	16	45.43

Table 2.2: Throughput (in MB/s) and latency (in microsecs) of internode point-to-point communication with `ncclSend`/`ncclRecv` with different message sizes, as reported by the `nccl-tests` benchmarks on 2 H100 GPUs on MareNostrum5. (Theoretical max bandwidth is 25 GB/s.)

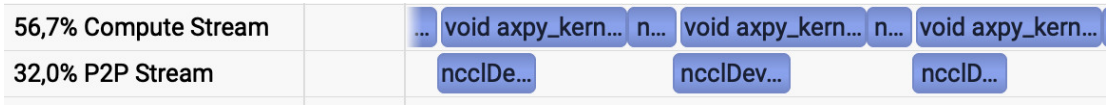
Size [B]	Throughput [MB/s]	Latency [µs]
1,024	47.4	21.59
2,048	95.6	21.42
4,096	189.3	21.63
8,192	369.0	22.20
16,384	648.6	25.26
32,768	1,198.1	27.35
65,536	1,982.9	33.05
131,072	3,557.9	36.84
262,144	6,358.1	41.23
524,288	9,271.2	56.55
1,048,576	12,979.0	80.79

Table 2.3: Throughput (in GB/s) as reported by the `BabelStream` benchmark on a MareNostrum 5 ACC node with an NVIDIA H100 64 GB GPU that has a theoretical peak memory bandwidth of 1,631.2 GB/s with active ECC.

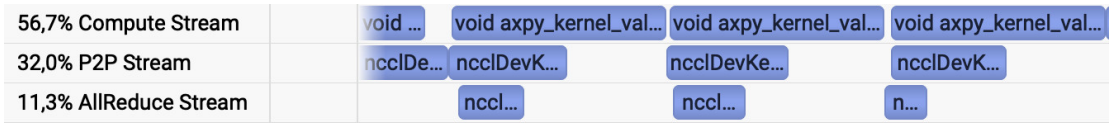
Kernel	Throughput [GB/s]	Effective bandwidth [%]
Copy	1,484.3	91.0 %
Mul	1,459.0	89.4 %
Add	1,509.7	92.5 %
Triad	1,514.3	92.8 %
Dot	1,529.8	93.7 %

remain scalable. To put it in context, Table 2.2 indicates that a single P2P message takes roughly the same amount of time when its size lies between 256 KiB and 512 KiB. Likewise, Table 2.3 reports that bandwidth-bound compute kernels such as *Add* and *Dot* sustain around 1,500 GB/s on a MareNostrum 5 H100 GPU, so in 45 μs they can process roughly 68.6 MB of data, corresponding to double-precision arrays of about 8.6 million entries. These figures give the sense of how much P2P communication or computation would be required to completely hide the collective communication that occurs in every Pipe-PCG iteration. They also make clear that whether the collective can be hidden depends on the problem, for sufficiently large partitions, there is plenty of independent P2P traffic and computation to overlap it with.

Based on this observation, we developed a custom three-stream microbenchmark that concurrently executes collective communication, P2P communication, and computation, in order to isolate and measure the effect of overlapping all three at once. The benchmark consists of an allreduce collective over two double-precision scalars (16 bytes), a send/recv operation arranged as a simple ring topology with a payload of 32,000 doubles (256 KiB), and a DAXPY compute kernel that processes 64 MB of data. The baseline for comparison is a traditional two-stream overlap strategy, in which a compute stream holds both the allreduce and the DAXPY kernel while a



a. Two-Stream Implementation



b. Three-Stream Implementation

Figure 2.1: NVIDIA Nsight Systems traces for the three-stream microbenchmark on 4 GPUs. **(a)** Two-stream baseline, in which the allreduce and DAXPY operations share the Compute stream alongside a separate P2P stream. **(b)** Proposed three-stream approach, showing clear concurrent overlap across dedicated Compute, P2P, and Allreduce streams.

separate communication stream handles the P2P transfers; that is, the DAXPY is overlapped with the P2P transfer, but the allreduce collective is executed without any overlap. Our proposed configuration instead dedicates a third stream to the allreduce, decoupling it from the compute stream so that all three operations can run concurrently. In order to avoid noise and outlier measurements, we ran both variants for 10^5 iterations.

On a single node with four GPUs, the three-stream configuration outperformed the two-stream baseline by approximately 5% in total execution time. While modest at this small scale, the improvement confirms that a third stream can extract overlap that the traditional two-stream strategy leaves unused, and the cost figures above suggest that the benefit grows as the exposed collective latency increases with GPU count. To verify that the speedup indeed stems from genuine concurrency rather than from measurement artefacts, we inspected the execution with the NVIDIA Nsight Systems profiler. As shown in Figure 2.1, the profiler traces exhibit clear overlap among all three streams, *compute*, *P2P*, and *allreduce*, in the three-stream design, whereas in the two-stream baseline the allreduce is serialized within the compute stream.

These microbenchmark results establish the main argument of this thesis, for a solver such as the ones in Krylov subspace solvers that issues both collective and P2P communication every iteration, there is real performance to be gained by overlapping the two communication types with each other and with computation, going beyond the traditional model that overlaps computation with a single communication stream. The challenge, addressed in Chapter 3, is to realize this triple overlap inside a full distributed Krylov solver, where the three streams must be coordinated correctly and where they contend for shared GPU and network resources.

Chapter 3

MULTI-GPU IMPLEMENTATION OF KRYLOV SUBSPACE SOLVERS

This chapter describes how we realize the pipelined preconditioned conjugate gradient (Pipe-PCG) and pipelined stabilized biconjugate gradient (Pipe-BICGSTAB) methods on distributed multi-GPU systems. We present two implementations. The first is a *two-stream* design that follows the state-of-the-art computation–communication overlap strategy of Trotter et al. [Trotter et al., 2025]; we use it as our baseline because it achieves state-of-the-art performance among the CG implementations surveyed in Chapter 6. The second is our proposed *three-stream* design, the main contribution of this thesis, which additionally overlaps the collective communication of the dot products with both computation and the point-to-point (P2P) halo exchange. Building on the observations of the three-stream microbenchmark in Chapter 2, where a dedicated stream for the collective operation produced a measurable speedup, we now realize that idea in a full solver and confront the engineering obstacles.

We implement both designs for NVIDIA GPUs using the NVIDIA Collective Communication Library (NCCL) [NVIDIA Corporation, 2026b] for inter-GPU communication. NCCL natively supports CUDA streams, which is the key enabler for issuing asynchronous, overlapping GPU operations without returning control to the host between kernels. Prior studies have shown that, on supercomputers, NCCL delivers performance comparable to GPU-aware MPI [Trotter et al., 2025, De Sensi et al., 2024], so it does not sacrifice raw communication throughput for this programmability. Section 3.4 discusses how the same design could instead be expressed with GPU-aware MPI or NVSHMEM, and why we ultimately based this work on NCCL.

3.1 Multi-GPU Pipelined PCG and Pipelined BICGSTAB

Executing Pipe-PCG and Pipe-BICGSTAB in a distributed-memory setting first require partitioning the sparse system matrix A and its associated vectors across the participating GPUs. We adopt a *row-wise* partitioning strategy, the rows of A are divided among the ranks, and each GPU receives its assigned block of rows together with the corresponding segments of the solution and auxiliary vectors. Once this distribution is fixed, each GPU owns a separate partition, and the remaining vector operations can proceed locally within each GPU's memory. The only operations that go beyond a single GPU's memory are the two communication patterns that we describe in Section 3.1.2, the global reduction required by the dot products, and the halo exchange required by the distributed sparse matrix-vector multiplication (SpMV).

The compute kernels themselves are the same fundamental building blocks introduced in Chapter 2, which are vector updates (AXPY), dot products (DOT), one SpMV, and the application of a preconditioner for Pipe-PCG. While a single Pipe-PCG iteration performs two combined dot products, eight AXPY operations, one SpMV, and one preconditioning step; Pipe-BICGSTAB performs seven dot products, eight AXPY operations, two SpMVs and the calculations of the scalars.

3.1.1 Computation

For GPU implementations of CG methods, the accepted best practice [Mills et al., 2024] is to offload the compute kernels to the device while retaining lightweight, inherently scalar work on the host, such as convergence checks and the computation of the scalar coefficients α and β (lines 8 and 10 of Algorithm 1 in Chapter 2). For the standard kernels, we rely on vendor-optimized libraries such as cuBLAS [NVIDIA Corporation, 2025] for the local dot products and cuSPARSE [NVIDIA Corporation, 2022] for the SpMV. cuSPARSE internally employs merge-based SpMV kernels [Merrill and Garland, 2016], which balance work across the rows of the matrix regardless of how irregular the nonzero distribution is. In our early experiments these kernels outperformed hand-written SpMV kernels, which is why we adopt them here.

While vendor libraries make the AXPY operations straightforward to express, using them naively introduces a subtle inefficiency that Trotter et al. [Trotter et al., 2025] identified. Each AXPY needs the scalar coefficients α and β, and if those scalars are computed on the device but consumed by a host-launched library call, the implementation must repeatedly move them between host and device and re-launch kernels around them. This back-and-forth exposes kernel-launch latency and host–device transfer overhead on the critical path. To eliminate it, we fuse all eight AXPY updates into a single custom compute kernel in Pipe-PCG and, compute α and β *on the device* inside that same kernel, as shown in the fused block of Listing 3.1. Similarly in Pipe-BICGSTAB, we fused eight AXPY operations into three separate kernels and, fused α and β scalars into a single kernel to avoid multiple CUBLAS kernel launch overheads and expensive host–device transfers as shown in the Listing 3.2. Fusing the updates also amortizes the cost of reading and writing the auxiliary vectors, since the fused kernel streams each vector through the GPU's memory hierarchy once rather than once per individual AXPY. Because the coefficients never leave the device, the host is not involved in the update phase at all, which is essential for keeping the streams busy without host synchronization.

For the preconditioner used in Pipe-PCG, we select the Jacobi preconditioner for its simplicity and very low computational overhead. It is defined as the diagonal of the system matrix, $M = \mathrm{diag}(A)$, and its inverse M^{-1} is obtained by taking the elementwise reciprocals of the diagonal entries. Applying M^{-1} is therefore a single elementwise vector scaling, which adds negligible arithmetic cost per iteration while still improving the conditioning of the system. Because a Jacobi preconditioner is not provided by the vendor libraries, we implement the application kernel ourselves. This also gives us direct control over its stream placement and its streaming-multiprocessor (SM) footprint, which matters for the resource-contention considerations discussed in Chapter 4.

```
1  Input: A^(p) = Ā^(p) + Â^(p), b^(p), x_0^(p), M, ϵ
2  r_0 ← b − Ax_0, u_0 ← M^{-1} r_0, w_0 ← Au_0
3  For i = 0, 1, 2, ... Do
4    GPU DOT:    γ_i^(p) ← r_i · u_i
5    GPU DOT:    δ^(p) ← w_i · u_i
6    SYNC:       Wait for dot products to complete
7    ALLREDUCE:  γ_i ← Σ_p γ_i^(p); δ ← Σ_p δ^(p) (non-blocking)
8    GPU Preconditioner:   m_i ← M^{-1} w_i
9    SYNC:       Wait until m_i is ready
10   P2P PACK:   Pack halo m̂_i for sending
11   P2P COMM:   Start sending halo m̂_i to remote GPUs
12   GPU SpMV:   n_i ← Ā^(p) * m_i (Diagonal)
13   SYNC:       Wait until m̂_i is received
14   GPU SpMV:   n_i ← n_i + Â^(p) * m̂_i (Off-diagonal)
15   SYNC:       Wait until allreduce is done
16   CPU:        If ||r_i|| ≤ ϵ||r_0|| Then Exit: Converged
17   GPU FUSED AXPY:
18         If i == 0: α_0 ← γ_0/δ, β_0 = 0
19         Else: α_i ← γ_i/(δ − β_i γ_i/α_{i−1}), β_i ← γ_i/γ_{i−1}
20         z_i ← n_i + β_i z_{i−1}
21         q_i ← m_i + β_i q_{i−1}
22         s_i ← w_i + β_i s_{i−1}
23         p_i ← u_i + β_i p_{i−1}
24         x_{i+1} ← x_i + α_i p_i
25         r_{i+1} ← r_i − α_i s_i
26         u_{i+1} ← u_i − α_i q_i
27         w_{i+1} ← w_i − α_i z_i
28 End For
```

Listing 3.1: Pseudo-code of the multi-GPU three-stream Pipe-PCG implementation. Colors indicate the stream on which each operation is executed: the *compute stream* in green, the *P2P stream* in purple, the *collective stream* in brown, and *host-side CPU operations* in blue.

1 Input: $\boldsymbol{A}^{(p)} = \bar{\boldsymbol{A}}^{(p)} + \hat{\boldsymbol{A}}^{(p)}$, $\boldsymbol{b}^{(p)}$, $\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$,

2 $\tilde{\boldsymbol{r}} \leftarrow \boldsymbol{r}_0$, $\boldsymbol{w}_0 \leftarrow \boldsymbol{r}_0$, $\boldsymbol{t}_0 \leftarrow \boldsymbol{w}_0$

3 $\rho_0 \leftarrow \boldsymbol{r}_0 \cdot \boldsymbol{r}_0$, $\alpha_0 \leftarrow 0/(\boldsymbol{r}_0 \cdot \boldsymbol{w}_0)$, $\beta_{-1} \leftarrow$,\omega_{-1} \gets 0$p_{-1} = \boldsymbol{s}_{-1} = \boldsymbol{z}_{-1} =_{-1} \leftarrow \boldsymbol{0}$

4 For $i = 0, 1, 2, \ldots$ Do

5 GPU FUSED AXPY:

6 $\boldsymbol{p}_i \leftarrow \boldsymbol{r}_i + i - 1(\boldsymbol{p}_{i-1} - \omega_{i-1}\boldsymbol{s}_{i-1})$

7 $\boldsymbol{s}_i \leftarrow \boldsymbol{w}_i + i - 1(\boldsymbol{s}_{i-1} - \omega_{i-1}\boldsymbol{z}_{i-1})$

8 $\boldsymbol{z}_i \leftarrow \boldsymbol{t}_i + i - 1(\boldsymbol{z}_{i-1} - \omega_{i-1}\boldsymbol{v}_{i-1})$

9 SYNC: Wait until $\boldsymbol{z}_i$ is ready

10 P2P PACK: Pack halo $\hat{\boldsymbol{z}}_i$ for sending

11 P2P COMM: Start sending halo $\hat{\boldsymbol{z}}_i$ to remote GPUs

12 GPU FUSED AXPY:

13 $\boldsymbol{q}_i \leftarrow \boldsymbol{r}_i - \alpha_i \boldsymbol{s}_i$,

14 $\boldsymbol{y}_i \leftarrow \boldsymbol{w}_i - \alpha_i \boldsymbol{z}_i$

15 GPU DOT: $\sigma_1^{(p)} \leftarrow \boldsymbol{q}_i \cdot \boldsymbol{y}_i$;

16 GPU DOT: $\sigma_2^{(p)} \leftarrow \boldsymbol{y}_i \cdot \boldsymbol{y}_i$

17 SYNC: Wait for dot products to complete

18 ALLREDUCE: $\sigma_k \leftarrow \sum_p \sigma_k^{(p)}$, $k = 1, 2$ (non-blocking)

19 GPU SpMV: $\boldsymbol{v}_i \leftarrow A^{(p)} * \boldsymbol{z}_i$ (Diagonal)

20 SYNC: Wait until $\hat{\boldsymbol{z}}_i$ is received

21 GPU SpMV: $\boldsymbol{v}_i \leftarrow_i + \hat{\boldsymbol{A}}^{(p)} * \hat{\boldsymbol{z}}_i$ (Off-diagonal)

22 SYNC: Wait until first allreduce is done

23 GPU FUSED AXPY:

24 $\omega_i \leftarrow \sigma_1/\sigma_2$

25 $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + i\boldsymbol{p}_i + \omega_i \boldsymbol{q}_i$

26 $\boldsymbol{r}_{i+1} \leftarrow \boldsymbol{q}_i - i\boldsymbol{y}_i$

27 $\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{y}_i - i(\boldsymbol{t}_i - \alpha_i \boldsymbol{v}_i)$

28 SYNC: Wait until $\boldsymbol{w}_{i+1}$ is ready

29 P2P PACK: Pack halo $\hat{\boldsymbol{w}}_{i+1}$ for sending

30 P2P COMM: Start sending halo $\hat{\boldsymbol{w}}_{i+1}$ to remote GPUs

31 GPU DOT: $\delta_1^{(p)} \leftarrow \tilde{\boldsymbol{r}} \cdot \boldsymbol{s}_i$;

32 GPU DOT: $\delta_2^{(p)} \leftarrow \tilde{\boldsymbol{r}} \cdot \boldsymbol{z}_i$

33 GPU DOT: $\delta_3^{(p)} \leftarrow \boldsymbol{r}_i \cdot \boldsymbol{w}_{i+1}$

34 GPU DOT: $\delta_4^{(p)} \leftarrow \boldsymbol{r}_i \cdot \boldsymbol{r}_{i+1}$

35 GPU DOT: $\eta^{(p)} \leftarrow \boldsymbol{r}_{i+1} \cdot \boldsymbol{r}_{i+1}$

36

```
SYNC:       Wait for dot products to complete
ALLREDUCE:  δ_k ← Σ_p δ_k^(p),  k = 1,...,4;  η ← Σ_p η^(p)  (non blocking)
SYNC:       Wait until the reduced values are readable
D2H COPY:   Stage η, ω_i and ρ_i in pinned host memory
GPU SpMV:   t_{i+1} ← Ā^(p) * w_{i+1}  (Diagonal)
SYNC:       Wait until ŵ_{i+1} is received
GPU SpMV:   t_{i+1} ← t_{i+1} + Â^(p) * ŵ_{i+1}  (Off-diagonal)
SYNC:       Wait until second allreduce is done and ρ_i is staged
GPU SCALARS:
        β_i ← (α_i/ω_i)(δ_1/ρ_i)
        α_{i+1} ← δ_1/(δ_2 + β_i δ_3 − i ω_i δ_4),  ρ_{i+1} ← δ_1
CPU:        ||r_{i+1}|| ← √η
CPU:        If ||r_{i+1}|| ≤ |r_0|| Then Exit: Converged
CPU:        If ω_i = 0 Or ρ_i = 0 Then Exit: Breakdown
CPU:        If i > 0 And i mod ν = 0 Then: (Residual Replacement in every 100
  iterations)
    GPU SpMV: r_{i+1} ← −A x_{i+1},  w_{i+1} ← A r_{i+1},  t_{i+1} ← A w_{i+1}
    GPU SpMV: s_i ← A p_i,  z_i ← A s_i,  v_i ← A z_i
End For
```

Listing 3.2: Pseudo-code of the multi-GPU four-stream pipelined BiCGStab implementation with periodic residual replacement. Colors indicate the stream on which each operation is executed: the *compute stream* in green, the *P2P stream* in purple, the *collective stream* in brown, the *scalar-copy stream* in red, and *host-side CPU operations* in blue.

3.1.2 Communication

Two distinct communication patterns arise when Pipe-PCG and Pipe-BICGSTAB solvers are distributed across GPUs, and they have very different characteristics.

The first is the global reduction behind the dot products. Each GPU can only compute a *partial* dot product over its locally owned vector segment. To recover the mathematically correct global scalar, these partial results must be summed across all GPUs with a collective `allreduce` operation. This communication is tiny in volume—only two or foru double-precision scalars, i.e., 16 bytes or 32-bytes—but it

is latency-bound and, in the pipelined formulation, occurs once per iteration in Pipe-PCG and twice per iteration in Pipe-BICGSTAB. Its cost grows with the number of participating GPUs rather than with the problem size, which is precisely why it becomes the dominant exposed cost at large scale, as the microbenchmarks in Chapter 2 indicate.

The second pattern concerns the distributed SpMV, $y = Ax$. Under row-wise partitioning, each GPU computes the entries of y for its assigned rows, but to do so it needs the entries of the input vector x that correspond to the nonzero columns of its matrix partition. Some of those columns fall outside the GPU's own row range, so their x values reside on remote GPUs. Fetching them generates an *irregular*, sparsity-dependent point-to-point communication pattern, a given GPU exchanges data only with the neighbors whose partitions supply the columns it references, and the number of such neighbors and the message sizes depend entirely on the matrix structure and the partitioning.

To hide the latency of this irregular exchange, we split each GPU's matrix partition into a *local* (or *diagonal*) component and a *remote* (or *off-diagonal*) component. The diagonal component contains the columns whose x values are already available locally, while the off-diagonal component contains the columns whose x values must be received from neighbors. This split enables a classic overlap, in which each GPU first multiplies the diagonal component with the locally available x values while simultaneously issuing the P2P transfers that fetch the missing remote values. Once those transfers complete, the GPU multiplies the off-diagonal component with the freshly received data and accumulates the result. In this way, the local diagonal SpMV is overlapped with the halo exchange, and this diagonal/off-diagonal decomposition is used consistently throughout both the two-stream and three-stream implementations. In practice the remote x values are gathered into a contiguous send buffer by a *pack* kernel before transfer and scattered into place by an *unpack* step on arrival, so that the P2P messages remain dense and the transfers efficient.

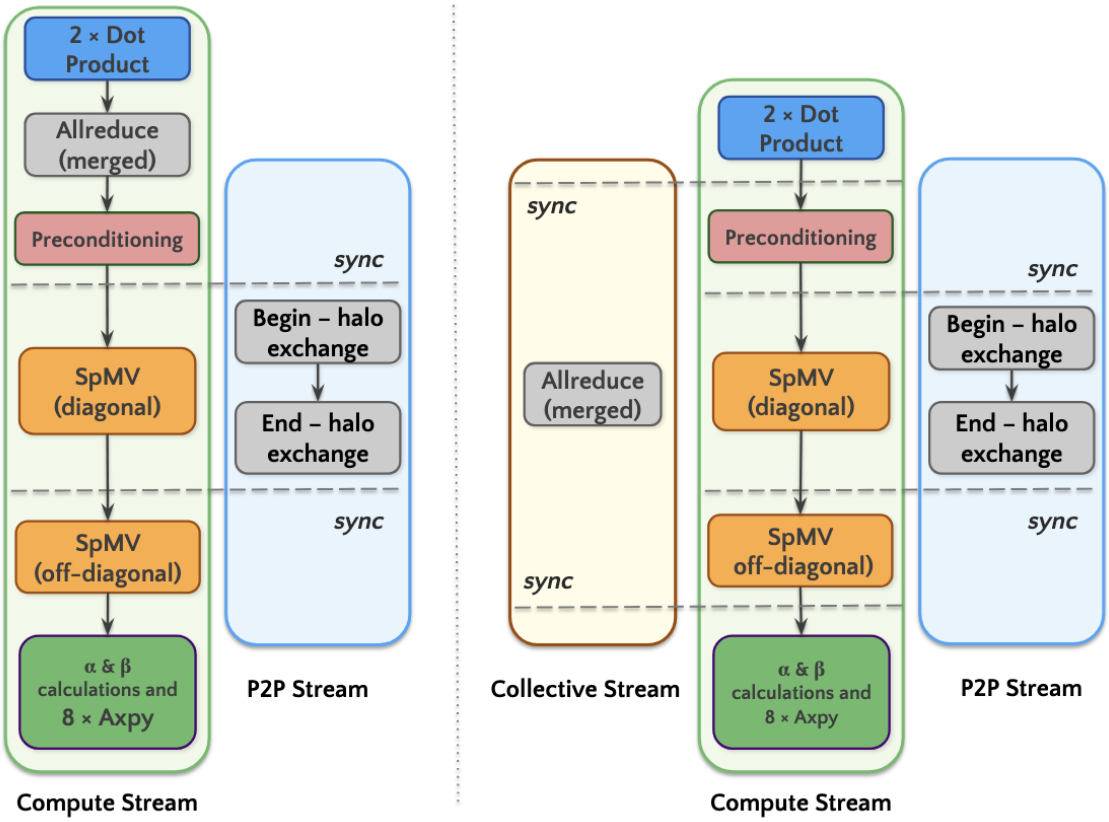


Figure 3.1: Two-stream versus three-stream design of pipelined preconditioned CG. The two-stream design overlaps the P2P halo exchange with computation, while the three-stream design additionally offloads the collective `allreduce` onto a dedicated stream so that it overlaps with both computation and P2P communication.

3.2 Two-Stream Baseline

The diagonal/off-diagonal decomposition of the SpMV exposes an opportunity to overlap the irregular P2P halo exchange with computation on the local part of the matrix. The two-stream design exploits exactly this opportunity using two CUDA streams, as illustrated on the left of Figure 3.1 and of Figure 3.2. Our implementation builds directly on the state-of-the-art approach of Trotter et al. [Trotter et al., 2025], which combines CUDA streams with NCCL specifically to avoid host-side synchronization.

Because NCCL is stream-aware, the GPU kernels that pack, send, receive, and

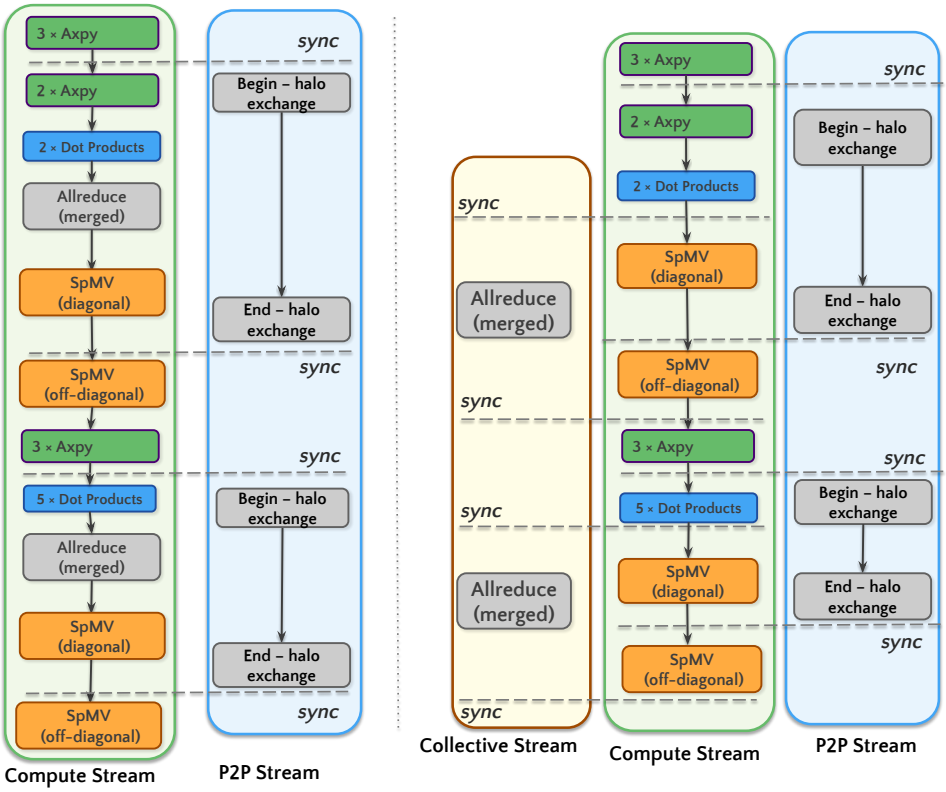


Figure 3.2: Two-stream versus three-stream design of pipelined BICGSAB. The two-stream design overlaps the P2P halo exchange with computation, while the three-stream design additionally offloads the collective `allreduce` onto a dedicated stream so that it overlaps with both computation and P2P communication.

unpack halo messages can be enqueued onto a stream asynchronously from the CPU without incurring any CPU–GPU synchronization. The dependencies between the *P2P* stream and the *compute* stream are then resolved entirely on the device through CUDA events, using `cudaEventRecord` to mark a point in one stream and `cudaStreamWaitEvent` to make another stream wait for it. This event-based coupling is what allows the two streams to be kept mutually consistent without calling `cudaStreamSynchronize` or otherwise returning to the host. Avoiding host synchronization matters for two reasons; first, a host-side wait would stall the CPU and prevent it from running ahead to enqueue subsequent kernels, so the GPU would sit idle between kernel launches while the host caught up; second, event-based waits

are resolved by the GPU's own scheduler, so they hide kernel-launch latency, which means as soon as the awaited event is reached, the dependent kernel begins without a round trip through the driver. Keeping the CPU out of the inner loop is therefore not a random detail but the mechanism that makes the overlap effective.

The resulting synchronization structure is shown in Figure 3.1 for Pipe-PCG and in Figure 3.2 for Pipe-BICGSTAB.

In Pipe-PCG, once the preconditioning step completes on the compute stream, we asynchronously record an event. We then make the P2P stream wait on this event. This synchronization guarantees that the preconditioned vector is fully computed before the P2P stream packs and transfers the halo data. Conversely, when the P2P stream finishes receiving the remote values, it records a second event. The compute stream waits on this event before launching the off-diagonal SpMV. Together, these synchronizations ensure data readiness for both the halo exchange and the off-diagonal multiplication, while allowing the diagonal SpMV to run concurrently with the transfer. We implement the non-blocking point-to-point communication using grouped ncclSend and ncclRecv calls enclosed by ncclGroupStart and ncclGroupEnd. This allows NCCL to fuse the many small neighbor exchanges of a single halo update into one efficient, batched operation.

We apply a similar approach in Pipe-BiCGSTAB. After the first group of fused AXPY operations completes on the compute stream, we signal an event. The P2P stream waits for this signal before packing the halo values and initiating communication. As soon as the P2P stream finishes transferring the halo data, it records another event. The compute stream waits on this event before launching the off-diagonal SpMV.

Despite this overlap, the two-stream design leaves one cost fully exposed. The collective Allreduce for the dot products runs on the compute stream, preventing it from overlapping with any other operation. The compute stream is busy executing the reduction, while the P2P stream handles only the halo exchange. As the GPU count grows and the Allreduce latency rises, this exposed collective becomes the primary obstacle to further scaling.

Although Figure 3.2 depicts the Allreduce collective and the halo exchange op-

erations as overlapping, in reality, they never run concurrently. This happens for two reasons. First, the halo exchange typically finishes long before the Allreduce collective begins. Second, NCCL strictly serializes distinct communication types in the background if they share a single communicator. Even in a hypothetical scenario where the halo exchange lasts longer and coincides with the Allreduce, NCCL would serialize them. The Allreduce collective would simply wait until the halo exchange concludes.

3.3 Three-Stream Approach

In the two-stream strategy, the collective communication required by the dot products remains a synchronization bottleneck. It cannot be overlapped with any other data transfer or computation. To remove this bottleneck from the critical path, we must overlap the collective operation concurrently with both local computation and the P2P halo exchange.

We achieve this triple overlap using a novel three-stream implementation of Pipe-PCG, shown on the right of Figure 3.1. We offload the collective operations to a dedicated Collective stream. The core mathematical kernels remain on the Compute stream, and the P2P halo transfers stay isolated on their own P2P stream. Listing 3.1 provides the full pseudo-code of the multi-GPU three-stream implementation, coloring each operation by the stream executing it. Reading top to bottom, the code launches the non-blocking Allreduce on the collective stream early—right after the local dot products. It progresses in the background while the compute and P2P streams execute the preconditioning, halo exchange, and SpMV. The compute stream only waits for the collective result at the very end of the iteration, just before the convergence check and the fused AXPY update.

We adopt a similar approach for the three-stream implementation of Pipe-BiCGSTAB, shown on the right side of Figure 3.2. We move the collective operations to the collective stream, while the rest remains identical to the two-stream implementation. However, isolating the collective operations on a separate stream introduces four extra synchronization points to the algorithm. Listing 3.2 shows the pseudo-

code for this implementation. Here, we overlap the collective Allreduce with both the SpMV and P2P communication by placing its synchronization point after the SpMV and before the fused AXPY operations. Similarly, we overlap the second collective Allreduce with the SpMV. Upon completion of this second Allreduce, we issue an asynchronous device-to-host copy for the convergence check. We overlap this asynchronous copy operation concurrently with the GPU scalar calculations.

However, simply placing the collective communication on a separate CUDA stream does not guarantee true overlap with the P2P transfers. Figure 3.3-a shows an NVIDIA Nsight Systems trace of a naive three-stream design that uses a single NCCL communicator for both collective and P2P operations. Even though three streams exist, only one communication operation actually overlaps with computation. The other remains queued until the active transfer finishes. Because NCCL strictly serializes communication operations issued from the same communicator, the collective and P2P transfers contend for the same resources and execute sequentially. In the trace, NCCL prioritizes the collective, delaying the P2P transfer until the Allreduce completes. This delay propagates downstream. Because the remote x values arrive late, the off-diagonal SpMV must execute much later than the diagonal SpMV, collapsing the intended overlap. This consequence is visible both in the profiler, which shows no genuine three-way overlap, and in the solver's total runtime, where this naive separation yields no speedup.

The root cause of this serialization is the shared communicator. To fix this, we must provide separate communicators for the two communication classes. We can achieve this in two ways.

First, we could create two fully independent NCCL communicators and assign one to each communication type. While this works, it forces the programmer to manually manage resources and avoid deadlocks. If we progress two independent communicators concurrently, they can deadlock if we enqueue their operations in incompatible orders across ranks. This occurs because each communicator enforces its own collective ordering that all ranks must respect.

Second, we can split a single existing communicator using the ncclCommSplit function. This approach partitions the parent communicator into sub-communicators

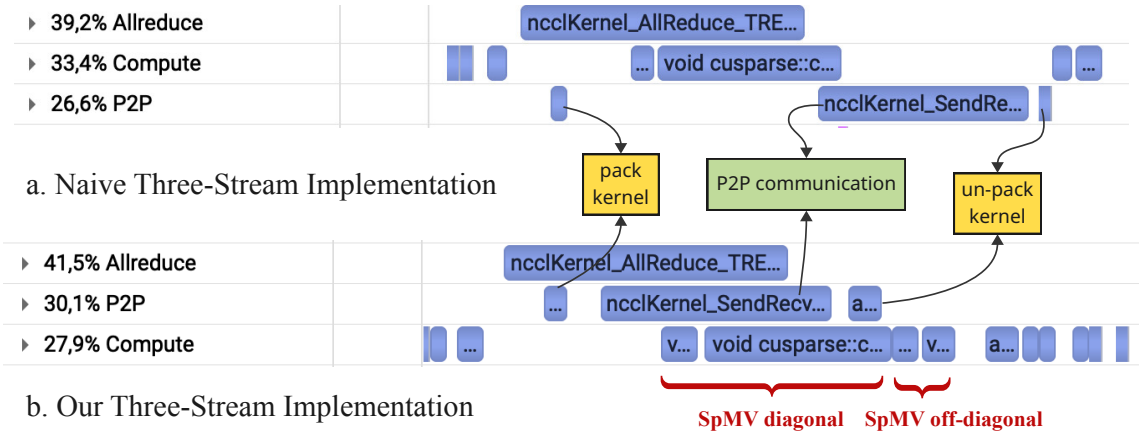


Figure 3.3: NVIDIA Nsight Systems profiling of the naive three-stream design versus our three-stream design. In (a), a single NCCL communicator serializes the collective and P2P operations, so no real overlap occurs; in (b), splitting the communicator with `ncclCommSplit` lets the collective, P2P, and compute streams run concurrently.

capable of executing different operations concurrently, which achieves the true overlap. We adopt `ncclCommSplit` because it handles the underlying resource management and deadlock avoidance internally. This allows the two communication streams to progress independently without requiring us to reason about cross-communicator ordering. Figure 3.3-b demonstrates the effect of this change. The Nsight Systems trace now exhibits genuine concurrent overlap across the compute, P2P, and collective streams.

3.4 Alternative Communication Libraries

Although we base our implementation on NCCL, the three-stream design is not inherently tied to it, and it is worth examining how the same overlap could be expressed with other communication libraries and why we did not adopt them.

In principle, an equivalent implementation can be written with GPU-aware MPI by using the non-blocking `MPI_Iallreduce` collective for the dot products and `MPI_Isend`/`MPI_Irecv` for the non-blocking P2P halo exchange. The difficulty is that support for non-blocking reductions on GPU-resident buffers is limited in some MPI

libraries, including OpenMPI [Community, 2026]. Where this support is missing, one must either stage the reduction data through CPU buffers—which reintroduces exactly the host–device transfers and host involvement that the streamed design was built to avoid—or tune the MPI implementation to route the collective through a specific GPU-capable backend such as HCOLL [NVIDIA Corporation, 2026a]. Both workarounds add complexity and, in the staging case, undermine the overlap, which makes GPU-aware MPI a poor fit for a design whose whole premise is keeping the collective asynchronous and on-device.

An alternative is NVSHMEM, where the collective can be performed on a separate stream directly with `nvshmem_double_sum_reduce_on_stream`. The P2P communication, however, must be expressed with one-sided operations, as described in the literature [Trotter et al., 2025, Sağbili et al., 2025]. This entails non-blocking Remote Memory Access (RMA) primitives such as `nvshmemx_double_put_on_stream`, combined with explicit signaling operations (e.g., `nvshmemx_double_put_signal_on_stream`) to guarantee correct synchronization and completion of the local and remote sides of each transfer. This one-sided model is expressive enough to realize the same triple overlap, but it shifts the burden of completion and ordering onto the programmer. We conducted preliminary testing and found that, for the message sizes and collective patterns relevant to our Pipe-PCG and Pipe-BICGSTAB solvers, NCCL's collective communication outperformed NVSHMEM. We therefore based this work solely on NCCL, while noting that the design itself is portable to these alternatives should the performance trade-offs change on other systems.

Chapter 4

ANALYTICAL COST MODEL AND PERFORMANCE IMPLICATIONS

The profiling traces in Chapter 3 demonstrate that the three-stream design achieves genuine concurrent overlap of computation, collective communication, and point-to-point (P2P) communication. Overlap by itself, however, does not guarantee a speedup, the benefit depends on how the individual costs of a Pipe-PCG or Pipe-BICGSTAB iteration relate to one another, and those relationships shift with the number of GPUs, the sparsity of the matrix, and the architecture of the machine. To reason about *when* the third stream pays off, we develop a simple analytical cost model of a single iteration of those solvers. The model is deliberately coarse—it counts the dominant per-stream costs rather than modeling every kernel exactly—but it is precise enough to expose the balance conditions that decide whether the three-stream design wins, ties, or is unnecessary.

The chapter is organized around the two solvers. Section 4.2 develops the model for Pipe-PCG, whose single reduction phase per iteration makes the structure of the argument easiest to see. Section 4.3 then extends the same framework to pipelined BiCGStab, where two reduction phases per iteration and the serialization imposed by a shared NCCL communicator change the analysis substantially. Section 4.4 closes with the resource contention effects that both models abstract away.

4.1 Notation

Throughout this chapter we use a common notation for the per-iteration costs incurred on one GPU. We write T_D for the local dot product cost, T_R for the collective `allreduce` cost, T_P for the preconditioner application, T_{SD} for the diagonal (local) SpMV, T_{SO} for the off-diagonal (remote) SpMV, T_H for the P2P halo exchange,

T_{Axpy} for the fused AXPY update, and T_{sync} for a single inter-stream synchronization. These are the same operations introduced in Chapter 3; the cost model simply assigns each a symbolic time and tracks which stream it lands on.

4.2 Analytical Cost Model for Pipelined PCG

4.2.1 Cost Model of Two-Stream Execution

In the two-stream design, computation and P2P communication execute on two separate streams. The P2P stream carries only the halo exchange needed by the distributed SpMV, whereas the compute stream is responsible for everything else—the local dot product, the collective `allreduce`, the preconditioner, both the diagonal and off-diagonal SpMV, the AXPY kernels, and the inter-stream synchronizations. Writing out the cost of each stream separately gives

$$T_{\text{2-stream}}^{(\text{comp})} = T_D + T_R + T_P + T_{SD} + T_{SO} + T_{\text{Axpy}} + 2T_{\text{sync}}, \tag{4.1}$$

$$T_{\text{2-stream}}^{(\text{P2P})} = T_H + 2T_{\text{sync}}. \tag{4.2}$$

The first equation simply sums the costs of the kernels that the compute stream executes in sequence. Two remarks make it easier to read. First, the collective term T_R appears *inside* the compute stream, in the two-stream design the `allreduce` has no stream of its own, so it is serialized with the rest of the compute work and is fully exposed. Second, the $2T_{\text{sync}}$ term accounts for the two event-based synchronization points that couple the compute and P2P streams within one iteration—one before the halo exchange, to ensure the vector to be sent is ready, and one after it, to ensure the received data is available before the off-diagonal SpMV. The P2P stream, by contrast, does almost nothing beyond the halo exchange itself, its cost is T_H plus the same two synchronization points, which it must also observe.

Because the two streams run concurrently, the wall-clock time of the parts that overlap is the *maximum* of the two streams, not their sum. The only pair of operations that genuinely overlaps here is the diagonal SpMV on the compute stream with the halo exchange on the P2P stream. All the other compute-stream work—the dot product, the exposed `allreduce`, the preconditioner, the off-diagonal SpMV, and

the AXPY—has nothing to overlap with and is therefore additive. Collecting the additive terms and replacing the overlapping pair with a maximum yields the total iteration cost:

$$T_{\text{2-stream}} \approx T_D + T_R + T_P + T_{SO} + T_{Axpy} + 2T_{\text{sync}} + \max(T_{SD},\, T_H). \tag{4.3}$$

The $\max(T_{SD}, T_H)$ term captures the overlap of diagonal SpMV with halo exchange, whichever of the two takes longer determines how long that overlapped phase lasts, and the shorter one is hidden entirely behind it. Everything else, including the exposed collective T_R, is paid in full. This last point is the central weakness of the two-stream design that the three-stream design sets out to remove.

4.2.2 Cost Model of Three-Stream Execution

In the three-stream design, computation, collective communication, and P2P communication each run on their own stream. The collective `allreduce` is lifted out of the compute stream and given a dedicated collective stream, which changes the per-stream accounting to

$$T^{(\text{comp})}_{\text{3-stream}} = T_D + T_P + T_{SD} + T_{SO} + T_{\text{Axpy}} + 4T_{\text{sync}}, \tag{4.4}$$

$$T^{(\text{coll})}_{\text{3-stream}} = T_R + 2T_{\text{sync}}, \tag{4.5}$$

$$T^{(\text{P2P})}_{\text{3-stream}} = T_H + 2T_{\text{sync}}. \tag{4.6}$$

Comparing the compute stream here with its two-stream counterpart, the T_R term has disappeared, the `allreduce` now lives on the collective stream and is no longer serialized with the compute kernels. The price of introducing that third stream is additional synchronization—the compute stream now carries $4T_{\text{sync}}$ rather than $2T_{\text{sync}}$, reflecting the extra event coordination needed to launch the collective early and to wait on its result before the AXPY update. The collective stream contributes T_R plus its own two synchronization points, and the P2P stream is unchanged from the two-stream case at $T_H + 2T_{\text{sync}}$.

Since the three streams execute concurrently, the total iteration time is governed by the longest of them rather than by their sum. Only the local dot product, the

AXPY update, and the two enclosing synchronizations are strictly serial—the dot product must finish before its result can be reduced, and the AXPY update depends on the reduced scalars—so those terms sit outside the maximum. Everything else is subject to the three-way overlap, giving

$$T_{\text{3-stream}} \approx T_D + T_{\text{Axpy}} + 2T_{\text{sync}} + \max\big(T_R, T_P + \max\big(T_{SD}, T_H\big) + T_{SO} + 2T_{\text{sync}}\big). \tag{4.7}$$

The outer maximum contrasts the two things that now proceed in parallel. Its first argument, T_R, is the entire cost of the collective stream. Its second argument is the cost of the "compute plus P2P" pipeline, the preconditioner T_P, followed by the overlapped diagonal-SpMV-versus-halo phase $\max(T_{SD}, T_H)$ inherited from the two-stream analysis, followed by the off-diagonal SpMV T_{SO} and the two synchronizations that bracket the halo exchange. In words, the iteration takes as long as whichever of these two concurrent tracks is slower. If the collective is the shorter of the two, it is hidden completely behind the compute-and-P2P work, and the solver pays nothing for it beyond the synchronization overhead—which is exactly the outcome the two-stream design could not achieve.

This structure makes the ideal operating point explicit. The best overlap is obtained when the collective stream is balanced against the compute stream, that is, when the two arguments of the outer maximum are approximately equal:

$$T_R \approx T_P + \max\big(T_{SD}, T_H\big) + T_{SO} + 2T_{\text{sync}}. \tag{4.8}$$

When this holds, neither track waits idle for the other, and the collective is almost entirely masked by useful work. A second, nested balance condition governs the inner overlap between the diagonal SpMV and the halo exchange, which is best when their costs are comparable:

$$T_H \approx T_{SD}. \tag{4.9}$$

If halo exchange and diagonal SpMV are of similar duration, then each hides the other and no fraction of either is left exposed inside the compute-and-P2P track. Together, these two conditions describe a well-balanced iteration in which all three

streams are kept busy and the total cost approaches the length of a single stream rather than the sum of all three.

4.2.3 Performance Implications

The primary value of our cost model is that its terms respond independently to different problem and machine parameters. Therefore, we can use the balance conditions as predictions to identify which workloads favor the three-stream design. First, we establish how each cost scales. The Allreduce time T_R scales with the number of GPUs, as a larger reduction tree takes longer to traverse. The preconditioner T_P, the halo exchange T_H, and the off-diagonal SpMV T_{SO} scale primarily with the number of nonzeros (nnz). These operations related to the remote and off-diagonal structure, which grows alongside nnz. Conversely, the diagonal SpMV T_{SD} and the fused AXPY T_{Axpy} scale mainly with the number of rows, because they process local dense-vector work. Finally, the synchronization overhead T_{sync} depends on platform properties; such as GPU architecture, driver version, and software stack—rather than the specific problem. Based on these scaling behaviors, we examine four distinct regimes.

Effect of GPU count. As the GPU count increases, T_R grows and can become a dominant cost. This hurts the two-stream design most, because there the `allreduce` is fully exposed on the critical path and its growth is paid in full every iteration. The three-stream design, in contrast, can hide all or part of T_R behind the compute-and-P2P track, so its advantage widens as the machine grows. The gain is largest when the balance condition $T_R \approx T_P + \max(T_{SD}, T_H) + T_{SO} + 2T_{\text{sync}}$ is met, since then the collective stream is almost completely masked. This is the regime in which the third stream is most clearly worthwhile.

Effect of row count and nnz. Two matrix properties shape the compute cost, the number of rows and the number of nonzeros. If the matrix is extremely sparse—many rows but very few nonzeros per row—then the iteration can become dominated by the AXPY kernels and local dot products, whose cost tracks the row count.

Because these components are not overlapped in either design (they are the strictly serial terms in both cost equations), the benefit of any overlap strategy is limited in this regime, and the two- and three-stream designs behave similarly. Conversely, if *nnz* grows while the row count stays fixed, the matrix becomes denser, with more nonzeros per row on average. The added cost then concentrates in T_P, T_{SO}, and T_H, while T_{SD} and T_{Axpy} grow far less. The balance therefore shifts toward P2P communication and off-diagonal work rather than diagonal computation. The three-stream design remains beneficial as long as this heavier P2P work stays comparable to the compute stream and can still be hidden behind the overlap. But if the growth in *nnz* drives T_H or T_{SO} to dominate strongly, the solver becomes communication-bound, the overlap can no longer absorb the excess, and the marginal value of the third stream shrinks.

Effect of P2P neighbor count. A related case arises when a rank must communicate with many neighbors, which depends on the sparsity structure and the matrix density discussed above. More neighbors mean a larger and more fragmented halo exchange, inflating T_H. Since the halo exchange overlaps only with the diagonal SpMV, and T_{SD} depends mainly on the row count, a matrix with high average *nnz* per row offers relatively less diagonal work to hide a large T_H behind, so the inner $\max(T_{SD}, T_H)$ overlap becomes skewed. Nevertheless, the three-stream design can retain an advantage even here, because it still overlaps the collective `allreduce`—a benefit the two-stream design never provides regardless of the P2P structure.

Conditions favoring two streams. The cost model also delimits when the third stream is *not* worth its overhead. Separating the collective onto its own stream is beneficial only if the overlap it exposes exceeds the extra synchronization it introduces. Comparing the two total-cost expressions, the additional synchronization of the third stream is $2T_{\text{sync}}$, so the two-stream design can be preferable when

$$T_R \leq 2T_{\text{sync}},$$

that is, when the `allreduce` is so cheap that hiding it saves less time than the added synchronization costs. In practice this occurs when the solver is strongly

compute-bound, when communication costs are negligible, or when one communication component dominates so heavily that it cannot be effectively hidden no matter how the streams are arranged. Outside these corner cases, the collective is large enough to amortize the extra synchronization, and the three-stream design is favored.

4.3 Analytical Cost Model for Pipelined BiCGStab

We now extend the cost model of Section 4.2 to the pipelined BiCGStab solver, whose two-stream and three-stream schedules are shown in Figure 3.2. As before, the model describes a single solver iteration on one device.

4.3.1 Structural Differences and Notation

Three structural properties distinguish this setting from Pipe-PCG and shape the entire analysis:

1. **Two reduction phases per iteration.** BiCGStab performs two matrix–vector products per iteration, each preceded by two and four dot products. A single iteration therefore issues *two* allreduce operations and *two* halo exchanges, whereas Pipe-PCG issues one of each.

2. **Fragmented AXPY blocks.** The eight AXPY operations cannot be fused into a single kernel, since the reductions of the two phases separate them. They are grouped into three fused kernels (3, 2 and 3 operations).

3. **Early launch of the halo exchange.** In both designs the vector whose halo must be communicated is produced by the first fused AXPY kernel of each phase, so the P2P stream is released before the local dot products are issued.

We index the two phases by $k \in \{1, 2\}$ and reuse the notation of Section 4.1: T_R, T_H, T_{SD}, T_{SO} and T_{sync} denote the costs of the merged allreduce, the halo exchange, the diagonal and off-diagonal SpMV, and a stream synchronization; T_D is the cost

of one merged pair of local dot products. For the vector updates we write $T^{(1a)}_{\text{Axpy}}$, $T^{(1b)}_{\text{Axpy}}$ and $T^{(2)}_{\text{Axpy}}$ for the three fused AXPY kernels, and set

$$T^{(1)}_{\text{Axpy}} = T^{(1a)}_{\text{Axpy}} + T^{(1b)}_{\text{Axpy}}, \qquad T_{\text{Axpy}} = T^{(1)}_{\text{Axpy}} + T^{(2)}_{\text{Axpy}}.$$

4.3.2 Communication Serialization and the Halo Overlap Window

A crucial constraint governs the two-stream design. Although the halo exchange and the allreduce are issued on two different CUDA streams, they are submitted to the same NCCL communicator, and NCCL serializes operations that share a communicator regardless of their stream assignment. The two communication operations therefore cannot proceed concurrently, the one submitted first executes, and the other waits for it to complete. Since the P2P stream is released immediately after the first fused AXPY kernel, the halo exchange is submitted first and the allreduce is delayed until the halo exchange finishes. This is precisely the behaviour observed in our profiles, where the two operations execute back to back rather than concurrently. It is the same mechanism that produces the degenerate schedule of Figure 3.3(a), and it is why communicator splitting is required for a genuine three-stream design.

The consequence for the cost model is that the two designs expose the halo exchange to different amounts of cover. Let $\Omega^{(k)}_2$ denote the compute-stream work issued between the halo launch and the allreduce, i.e. the work that can hide the halo in the two-stream design:

$$\Omega^{(1)}_2 = T^{(1b)}_{\text{Axpy}} + T_D, \qquad \Omega^{(2)}_2 = T_D. \tag{4.10}$$

The diagonal SpMV does *not* appear here, it is issued after the allreduce on the compute stream, and the allreduce is itself blocked by the halo exchange, so no part of the SpMV can overlap with the halo.

In the three-stream design the communicator is split, the two communication operations proceed independently, and the allreduce no longer occupies the compute stream. The halo exchange is then covered additionally by the diagonal SpMV, giving

$$\Omega^{(k)}_3 \;=\; \Omega^{(k)}_2 + T_{SD}. \tag{4.11}$$

Writing

$$h_X^{(k)} \;=\; \max\bigl(0,\; T_H - \Omega_X^{(k)}\bigr), \qquad X \in \{2,3\}, \tag{4.12}$$

for the exposed halo cost, we have $h_3^{(k)} \le h_2^{(k)}$ for every T_H, with

$$h_2^{(k)} - h_3^{(k)} \;=\; \min\bigl(h_2^{(k)},\, T_{SD}\bigr). \tag{4.13}$$

The third stream thus improves performance through two independent mechanisms: it removes the allreduce from the compute stream, and it removes the serialization that prevents the halo exchange from overlapping the diagonal SpMV. Both are consequences of communicator splitting.

Note also that $\Omega_X^{(1)} > \Omega_X^{(2)}$, the second phase has no intermediate AXPY kernel, so its halo exchange is covered by less work and is the first to become exposed in either design.

4.3.3 Cost Model of Two-Stream Execution

The compute stream carries the local dot products, the allreduce, both parts of the SpMV, the AXPY kernels and the inter-stream synchronizations, and stalls for $h_2^{(k)}$ while the communicator is occupied by the halo exchange:

$$T_{\text{2-stream}}^{(\text{comp},k)} = T_{\text{Axpy}}^{(k)} + T_D + h_2^{(k)} + T_R + T_{SD} + T_{SO} + 2T_{\text{sync}}, \tag{4.14}$$

$$T_{\text{2-stream}}^{(\text{P2P},k)} = T_H + 2T_{\text{sync}}. \tag{4.15}$$

The compute stream is the critical path, so

$$T_{\text{2-stream}}^{(k)} \approx T_{\text{Axpy}}^{(k)} + T_D + h_2^{(k)} + T_R + T_{SD} + T_{SO} + 2T_{\text{sync}}, \tag{4.16}$$

and, summing over the two phases,

$$T_{\text{2-stream}} \approx T_{\text{Axpy}} + 2T_D + 2T_R + 2T_{SD} + 2T_{SO} + \sum_{k=1}^{2} h_2^{(k)} + 4T_{\text{sync}}. \tag{4.17}$$

Both allreduce operations remain fully exposed on the critical path, and the total communication cost of a phase, $\max(T_H, \Omega_2^{(k)}) + T_R$, is strictly serial, the two communication types can only be hidden behind computation, never behind each other.

4.3.4 Cost Model of Three-Stream Execution

With a split communicator, the allreduce of each phase runs on a dedicated collective stream concurrently with the halo exchange and the computation:

$$T_{\text{3-stream}}^{(\text{comp},k)} = T_{\text{Axpy}}^{(k)} + T_D + T_{SD} + h_3^{(k)} + T_{SO} + 4T_{\text{sync}}, \tag{4.18}$$

$$T_{\text{3-stream}}^{(\text{coll},k)} = T_R + 2T_{\text{sync}}, \tag{4.19}$$

$$T_{\text{3-stream}}^{(\text{P2P},k)} = T_H + 2T_{\text{sync}}. \tag{4.20}$$

The collective stream is released after the dot products and must complete before the AXPY kernel of the following phase, so it overlaps with the diagonal SpMV, any residual halo stall, and the off-diagonal SpMV. The phase completes when the slower of the compute and collective streams finishes:

$$T_{\text{3-stream}}^{(k)} \approx T_{\text{Axpy}}^{(k)} + T_D + 2T_{\text{sync}} + \max\Big(T_R,\ T_{SD} + h_3^{(k)} + T_{SO} + 2T_{\text{sync}}\Big), \tag{4.21}$$

so that

$$T_{\text{3-stream}} \approx T_{\text{Axpy}} + 2T_D + 4T_{\text{sync}} + \sum_{k=1}^{2} \max\Big(T_R,\ T_{SD} + h_3^{(k)} + T_{SO} + 2T_{\text{sync}}\Big). \tag{4.22}$$

The best overlap is expected when the collective stream is balanced against the compute stream of the same phase,

$$T_R \approx T_{SD} + h_3^{(k)} + T_{SO} + 2T_{\text{sync}}, \tag{4.23}$$

and when the halo exchange is fully covered by the work issued after its launch,

$$T_H \le \Omega_3^{(k)} = \Omega_2^{(k)} + T_{SD}. \tag{4.24}$$

Both conditions must hold twice per iteration, and by (4.10) the second is strictly harder to satisfy in the second phase.

4.3.5 Gain of the Third Stream

Subtracting (4.21) from (4.16) gives the benefit of one phase:

$$T_{\text{2-stream}}^{(k)} - T_{\text{3-stream}}^{(k)} \;=\; \min\Big(\underbrace{h_2^{(k)} + T_{SD} + T_{SO}}_{\text{reduction-bound ceiling}},\ \underbrace{T_R - 2T_{\text{sync}} + \min\big(h_2^{(k)}, T_{SD}\big)}_{\text{halo-bound ceiling}}\Big).$$

(4.25)

The two arguments are the savings in the two cases of (4.21). If the collective stream is the slower one, the saving is everything now hidden behind the allreduce, the halo stall and both SpMVs. If the compute stream is slower, it is the allreduce removed from that stream, less the extra synchronization, plus the part of the halo stall the diagonal SpMV can absorb.

Reduction-bound ceiling When the allreduce is large, the saving stops growing at $h_2^{(k)} + T_{SD} + T_{SO}$, there is no more work left to hide behind it. With the halo fully covered ($h_2^{(k)} = 0$) the best possible gain over an iteration is $2(T_{SD} + T_{SO})$, its entire SpMV time.

Halo-bound ceiling When T_H is large, $h_2^{(k)}$ and $h_3^{(k)}$ both grow, but their difference never exceeds T_{SD}. Therefore, the time savings stop at $T_R - 2T_{\text{sync}} + T_{SD}$ per phase. Since the overall execution time of both designs keeps increasing, the relative speedup gradually diminishes until the two designs perform identically.

Condition favouring two streams The gain is negative only when

$$T_R + \min\left(h_2^{(k)}, T_{SD}\right) \;\leq\; 2T_{\text{sync}}, \tag{4.26}$$

that is, when both benefits together cannot pay for the extra synchronization. With the halo hidden this reduces to the Pipe-PCG condition $T_R \leq 2T_{\text{sync}}$, and (4.25) becomes $2\min(T_{SD} + T_{SO},\, T_R - 2T_{\text{sync}})$ per iteration. If the serial work $T_{\text{Axpy}} + 2T_D$ is also negligible, the ratio $T_{\text{2-stream}}/T_{\text{3-stream}}$ approaches 2, the same theoretical maximum of 100% improvement in Pipe-PCG.

4.3.6 Performance Implications

Effect of GPU count T_R grows with the number of GPUs while T_{SD}, T_{SO} and $\Omega_X^{(k)}$ do not. By (4.25) the benefit therefore grows monotonically with scale until the reduction-bound ceiling is reached. Since the two-stream design exposes $2T_R$ per iteration rather than one, its cost degrades roughly twice as fast with scale as

the Pipe-PCG baseline, and we expect the three-stream advantage for BiCGStab to become visible at smaller GPU counts.

Effect of rows and nonzeros T_H and T_{SO} scale primarily with the number of nonzeros and the partitioning-induced neighbour count, while T_{SD} and T_{Axpy} scale with the number of rows. Increasing nnz at constant row count raises T_H faster than $\Omega_X^{(k)}$, increasing the serialization stall $h_2^{(k)}$ and hence the absolute benefit of the third stream, but eventually driving the solver into the halo-bound regime where the relative benefit decays. Very sparse matrices with many rows shift the iteration toward the non-overlappable AXPY and dot-product work and limit the benefit of either overlap strategy.

4.4 Resource Contention

The cost models above treat the per-stream times as fixed, but running three streams concurrently can inflate those times through resource contention. While the three-stream design maximizes overlap in principle, all three streams must share the same physical GPU and the same network interface, so they compete for streaming multiprocessors (SMs) and for network bandwidth. We consider each form of contention in turn.

SM contention. NCCL moves data using GPU kernels, and those kernels consume SM resources that would otherwise be available to the compute kernels. As Hu et al. [Hu et al., 2026] describe, a collective operation is divided into one or more communication channels, each occupying one SM. A single SM is typically sufficient for a small `allreduce`, but our P2P halo exchange can involve larger data volumes and multiple concurrent messages, and so it can benefit from being given several SMs. NCCL exposes control over this allocation through the `ncclConfig_t` structure. In our implementation we set `maxCTAs` to four for the P2P communicator and to two for the collective communicator, which limits them to at most four and two cooperative thread arrays (CTAs), respectively, so that P2P—the heavier consumer—may claim more SMs than the lightweight collective. These values are

only hints, NCCL may override them according to its internal heuristics, so they are not always enforced strictly. Managing the compute side raises a symmetric difficulty. We can cap the SM usage of our own custom kernels, such as the preconditioner and the fused AXPY, to leave room for communication, but we cannot do the same for the vendor-provided SpMV and DOT kernels. Fortunately, those kernels are memory-bandwidth bound rather than SM bound, so surrendering a few SMs to communication is not expected to noticeably degrade their performance.

Network contention. The three-stream design also increases network pressure by issuing the collective and P2P transfers concurrently. In our setting, however, their bandwidth demands are highly asymmetric. The `allreduce` moves only 16 bytes—two double-precision scalars—and contributes negligible bandwidth overhead regardless of the GPU count. The P2P halo exchange dominates the network traffic, and its volume depends on the matrix sparsity pattern and the partitioning. Because the collective consumes so little bandwidth, the two transfers rarely saturate the link at the same time, and although some contention is inevitable, the three-stream design still amortizes its cost effectively, especially at larger scales and for denser matrices where there is more compute and P2P work to hide the collective behind. In either case, the three-stream design improves performance or, at worst, matches the two-stream implementation. The experiments in Chapter 5 confirm this behavior.

Chapter 5

EXPERIMENTS AND EVALUATION

In this chapter, we evaluate the performance of the two-stream and three-stream implementations of pipelined preconditioned CG (Pipe-PCG) and pipelined stabilized biconjugate gradient (Pipe-BiCGSTAB), which we introduced in Chapter 3. Our goal is to determine whether the concurrency of the three-stream design—scheduling the collective reduction, P2P halo exchange, and local computation on separate streams translates into measurable speedups on production supercomputers. We also verify whether our observations align with the analytical cost model developed in Chapter 4.

We compare our three-stream implementation against two baselines. The first is the state-of-the-art two-stream implementation by Trotter et al. [Trotter et al., 2025]. This baseline overlaps the halo exchange with computation but leaves the collective reduction on the compute stream. The second baseline comprises the Pipe-PCG and Pipe-BiCGSTAB solvers shipped with PETSc 3.21.5. While our two stream-based variants rely on NCCL for inter-GPU communication, PETSc uses CUDA-aware MPI. Comparing against both baselines allows us to separate the benefits of our scheduling strategy from the benefits of the underlying communication library.

5.1 Experimental Setup

We ran all experiments on two large NVIDIA-based systems: MareNostrum 5 and Leonardo. We used identical solver codes, partitioning schemes, and measurement protocols across both machines. This section describes the hardware, software stack, test matrices, and measurement methodology.

5.1.1 Supercomputers

We selected two systems with a similar overall node architecture: four NVIDIA data-center GPUs per node, connected via NVLink and an InfiniBand fabric. However, they differ in GPU generation, memory bandwidth, and network bandwidth. Evaluating on both systems allows us to observe how the balance between compute and communication impacts the three-stream design, as predicted by our cost model in Chapter 4.

MareNostrum 5. Each node in the Accelerated (ACC) partition of the MareNostrum 5 supercomputer contains two 40-core Intel Xeon Platinum 8460Y+ CPUs and four NVIDIA H100 GPUs. Each GPU provides 64 GB of HBM2e memory. NVLink 4.0 connects all intra-node GPU pairs with a bi-directional bandwidth of 300 GB/s. Additionally, each GPU attaches to an NVIDIA ConnectX-7 network interface card (NIC) providing a unidirectional bandwidth of 200 Gb/s. The global network utilizes NDR InfiniBand in a three-layer fat-tree topology [Banchelli et al., 2025]. We scale our experiments on this machine up to 64 nodes (256 GPUs).

Leonardo. Each node in the Booster partition of the Leonardo supercomputer includes one 32-core Intel Xeon Platinum 8358 CPU and four non-standard NVIDIA A100 SXM4 GPUs. These specific GPUs feature 64 GB of memory with a bandwidth of 1640 GB/s, differing from standard A100 SXM4 models [Turisini et al., 2023]. Inside a node, NVLink 3.0 connects the GPUs, providing 200 GB/s of bandwidth between each pair. Each node also includes two dual-port Mellanox HDR100 ConnectX-6 InfiniBand NICs, supplying a total inter-node bandwidth of 400 Gb/s. Compared to MareNostrum 5, Leonardo combines an older GPU generation and lower network bandwidth with similar memory capacity. As we demonstrate later, this hardware difference shifts the compute-to-communication balance, directly affecting the performance gains of the three-stream design.

5.1.2 Software Stack

On MareNostrum 5, we use the NVIDIA HPC SDK 25.1, CUDA 12.6, HPC-X MPI 2.21, and NCCL 2.18.5. On Leonardo, we use the NVIDIA HPC SDK 24.3, CUDA 12.3, HPC-X MPI 2.17.1, and NCCL 2.18.5. We use PETSc 3.21.5 for our MPI baseline. To partition the matrices, we use METIS 5.1.0, ensuring each GPU receives a balanced share of rows and nonzeros. We also apply the environment flags recommended by De Sensi et al. [De Sensi et al., 2024] to maximize communication performance. While our two-stream and three-stream implementations use NCCL for all communication, PETSc relies entirely on MPI. Maintaining a consistent NCCL version across both machines and using a single MPI baseline controls our comparison. This ensures that any reported differences stem from the scheduling strategy and hardware, rather than library variations.

5.1.3 Test Matrices

We use two groups of test matrices, summarized in Table 5.1. The first group comes from the SuiteSparse Matrix Collection [Davis and Hu, 2011]. We selected its six largest symmetric positive-definite real-valued matrices. These widely used instances span multiple application domains and provide enough complexity to stress a multi-GPU CG solver. However, even the largest matrix in this set, Queen_4147 (4.1 million rows and 317 million nonzeros), lacks sufficient work to scale efficiently beyond 64 GPUs. Beyond that threshold, kernel-launch and communication overheads dominate because the per-GPU workload becomes too small. In order to observe the strong-scaling regime our three-stream design targets, and to evaluate scaling up to 256 GPUs, we include three extremely large test instances.

The first large instance, poisson3da_12, originates from discretizing a 3D Poisson equation on a unit-cube geometry. With roughly 144 million rows and 2.2 billion nonzeros, it is more than an order of magnitude larger than the SuiteSparse matrices. The other two large instances, heart15 and heart12, come from a 3D cardiac monodomain simulation of the human heart at two different mesh resolutions [Martinez-Navarro et al., 2019]. Built on unstructured tetrahedral meshes, these

Table 5.1: Sparse matrices used in experiments.

Matrix	Rows	Nonzeros	Nonzeros/Row
Queen_4147	4,147,110	316,548,962	76.33
Bump_2911	2,911,419	127,729,899	43.87
Flan_1565	1,564,794	114,165,372	72.96
audikw_1	943,695	77,651,847	82.28
Serena	1,391,349	64,131,971	46.09
Geo_1438	1,437,960	60,236,322	41.89
poisson3da_12	144,515,663	2,239,624,249	15.5
heart15	874,424,445	7,185,411,541	8.2
heart12	139,589,939	2,144,835,461	15.4

matrices are very sparse. For instance, heart15 contains about 874 million rows but averages only 8.2 nonzeros per row. Therefore, almost all of its cost stems from memory-bound SpMV operations, generating high communication volume relative to local arithmetic. Table 5.1 lists each instance alongside its row count, nonzero count, and average nonzeros per row.

5.1.4 Measurement Methodology

We measure execution time using `clock_gettime` on the host CPU for each MPI process. To ensure accurate and reproducible results, every solver first executes 10 warmup iterations. This brings the GPUs, caches, and NCCL communicators into a steady state. Following the warmup, we execute an MPI barrier and a CPU-GPU synchronization `cudaDeviceSynchronize` before starting the timer. This ensures all processes begin timing simultaneously and prevents residual warmup tasks from skewing the measurements. We perform a final CPU-GPU synchronization before stopping the timer, guaranteeing every GPU task completes before we record the elapsed time. Since processes may finish at slightly different times, we report the maximum execution time across all ranks to reflect the true wall-clock cost of the

collective solve.

To have a fair comparison, we evaluate all solvers (including PETSc) within the same job submission for a given matrix and GPU count. This setup ensures every solver processes an identical matrix partition on the exact same hardware, eliminating run-to-run placement effects. We repeat each measurement three times and report the average alongside error bars showing the variance, verifying the stability of each configuration. To avoid distortion from outlier iterations, we run the Pipe-PCG solver for 10000 iterations and the Pipe-BiCGSTAB solver for 1000 iterations, reporting the average of solver time. Finally, for our in-depth analysis in Section 5.3, we complement these end-to-end timings with fine-grained GPU kernel measurements using `cudaEventRecord`. This allows us to separate the computational cost of each phase from communication and synchronization overheads.

5.2 Results

We first present results for the SuiteSparse matrices before analyzing the three large instances. Figure 5.1 reports the SuiteSparse results on MareNostrum 5, and Figure 5.2 shows the corresponding results on Leonardo. In both figures, the bars display the solver time for PETSc, the two-stream implementation by Trotter et al. [Trotter et al., 2025], and our proposed three-stream design across increasing GPU counts. The numbers above the three-stream bars indicate its speedup over both the two-stream baseline and PETSc.

In all cases, our three-stream design outperforms the two-stream baseline. This performance gap widens consistently as the GPU count grows—from 2–6% at 4 GPUs up to 18–47% at 64 GPUs. The only exception is Geo_1438 on Leonardo, which exhibits unstable results (as shown by the large variability in its error bars). This widening gap directly confirms the predictions of our cost model in Chapter 4. As the GPU count increases, the collective Allreduce issued in every iteration becomes more expensive. Consequently, the three-stream design can hide a growing amount of collective time behind the P2P halo exchange. The two-stream design only overlaps the reduction with local computation; our three-stream design additionally

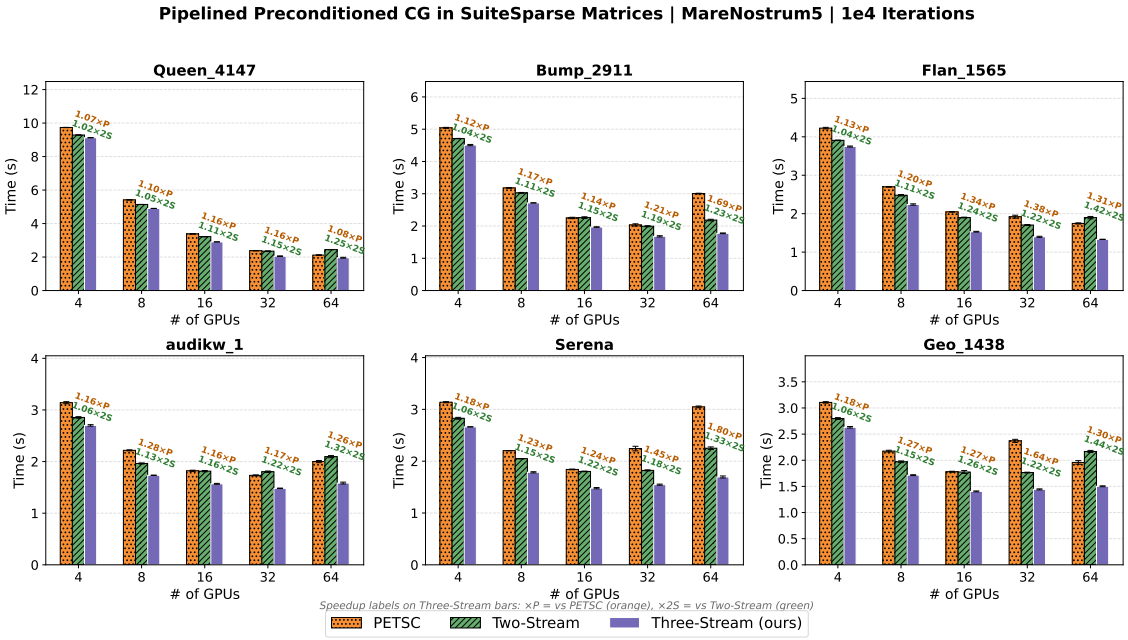


Figure 5.1: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the SuiteSparse matrices on the MareNostrum5 supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation.

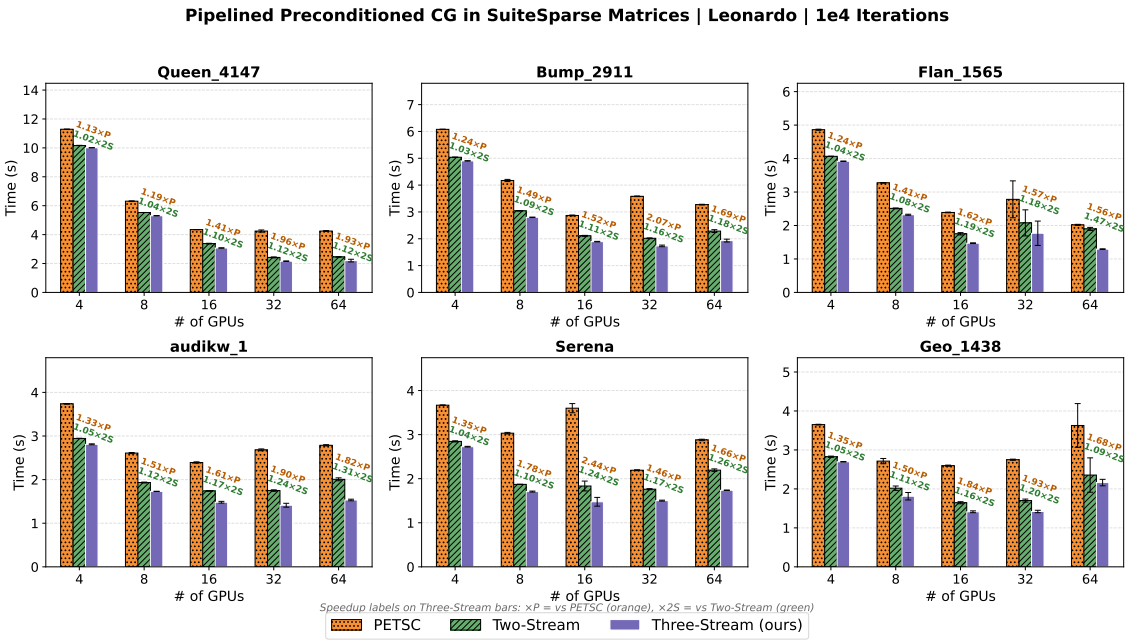


Figure 5.2: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the SuiteSparse matrices on the Leonardo supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation.

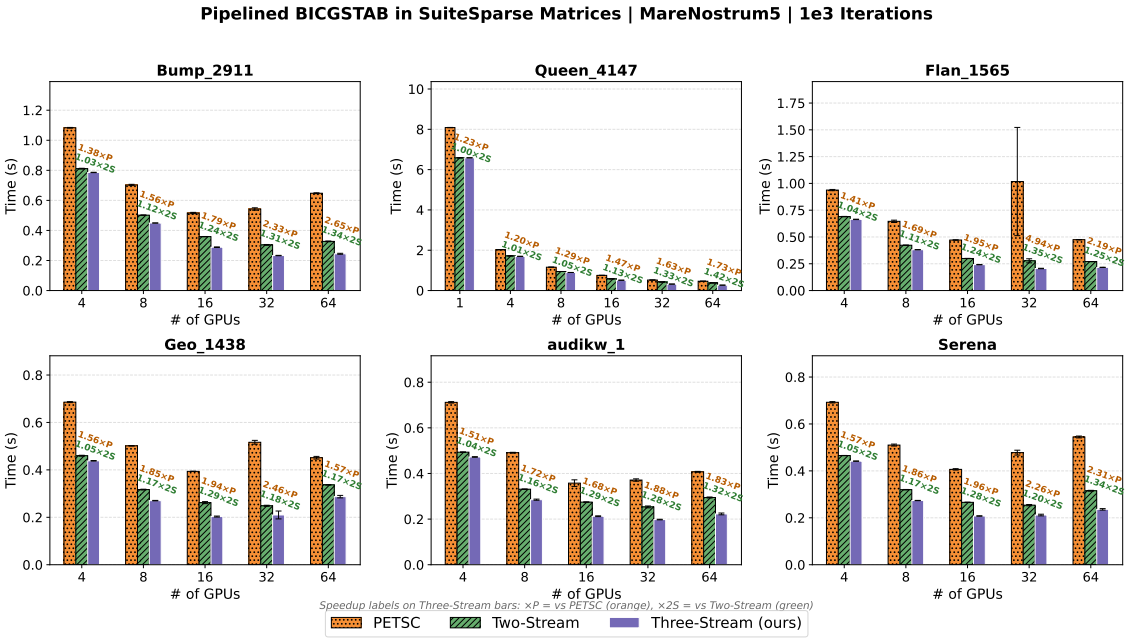


Figure 5.3: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the SuiteSparse matrices on the MareNostrum5 supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation and PETSc's Pipe-BICGSTAB implementation.

Table 5.2: Speed-ups of three-stream (ours) over PETSc and two-stream on MareNostrum5 (H100) and Leonardo (A100) using 64 GPUs in Pipe-PCG.

	MareNostrum5 (H100)		**Leonardo (A100)**	
Matrix	**vs. PETSc**	**vs. 2-stream**	**vs. PETSc**	**vs. 2-stream**
Queen_4147	1.08×	1.25×	1.93×	1.12×
Bump_2911	1.69×	1.23×	1.69×	1.18×
Flan_1565	1.31×	1.42×	1.56×	1.47×
audikw_1	1.26×	1.32×	1.82×	1.31×
Serena	1.80×	1.33×	1.66×	1.26×
Geo_1328	1.30×	1.44×	1.68×	1.09×
Geomean	1.38	1.33	1.72	1.23

overlaps it with the halo exchange. This extra concurrency becomes increasingly valuable as reduction costs rise. We note that the relative benefit naturally appears larger when the total runtime is smaller. At 64 GPUs, the runtimes for these SuiteSparse instances converge to roughly 1.5 seconds, making it difficult to observe clear separation among the matrices at that scale—a point we revisit in Section 5.3.

In addition to the figures, Table 5.2 reports the speedups of our three-stream design over both baselines at 64 GPUs. While PETSc (which relies on GPU-aware MPI rather than NCCL) is almost always slower than the stream-based implementations, this difference is more visible on Leonardo. There, our design is up to 93% faster than PETSc (a $1,93\times$ speedup on Queen_4147), with a geometric mean of $1,72\times$. In contrast, the speedup over the two-stream baseline is much lower, averaging $1,23\times$. On MareNostrum 5, these two speedups are much closer: a $1,38\times$ geometric mean over PETSc versus $1,33\times$ over the two-stream design. This indicates that on MareNostrum 5, PETSc and the two-stream implementation perform similarly. The large performance gap for PETSc on Leonardo aligns with that machine's

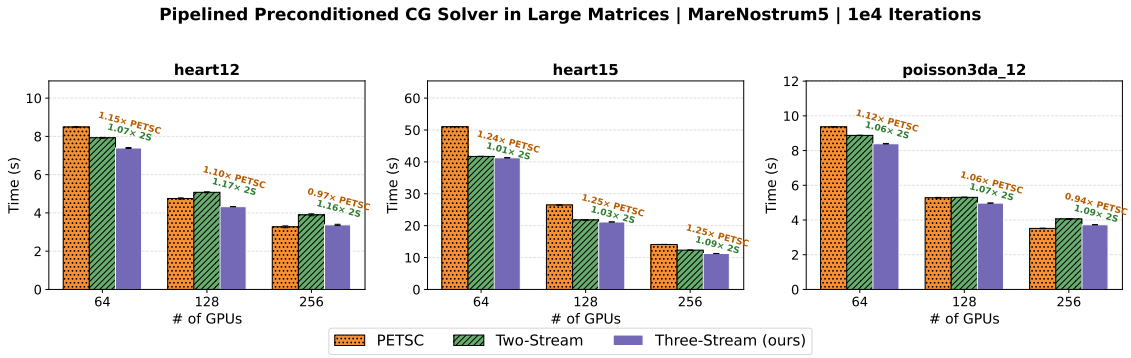


Figure 5.4: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the larger matrices on the MareNostrum5 supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation.

lower network bandwidth. On a more communication-bound system, the choice of communication library and the degree of overlap matter significantly more, allowing NCCL-based stream implementations to outperform the MPI baseline.

Regarding strong scaling, all matrices initially show performance improvements as we add GPUs. However, only the largest instance, Queen_4147, maintains near-linear scaling. None of the SuiteSparse instances show noticeable improvement when scaling to 64 GPUs. This implies we have reached the strong-scaling limit for these matrices, too little work remains per GPU for additional devices to provide a benefit. This observation directly motivates our use of the three large test instances.

Figure 5.5 shows the results for these larger instances on MareNostrum 5. Here, we observe that the relative benefit of the three-stream design is lower than on the SuiteSparse matrices. Because the per-GPU workload is much larger, communication accounts for a smaller fraction of the total time, leaving less overhead to hide. On these instances, PETSc is highly competitive and even outperforms the two-stream approach in several cases. Although system administrative policies prevented us from running even larger experiments, the speedups we observed at 256 GPUs strongly suggest that the three-stream design will continue to perform exceptionally well at higher GPU counts, where the communication fraction—and the

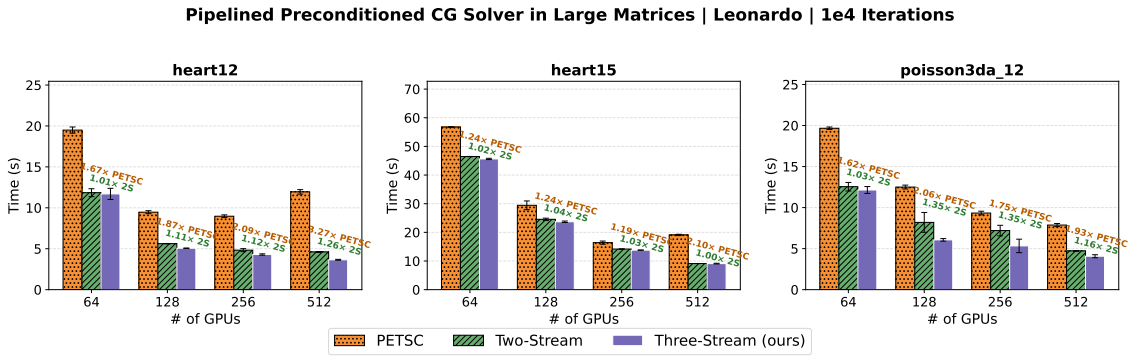


Figure 5.5: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the larger matrices on the Leonardo supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation and PETSc's Pipe-PCG implementations.

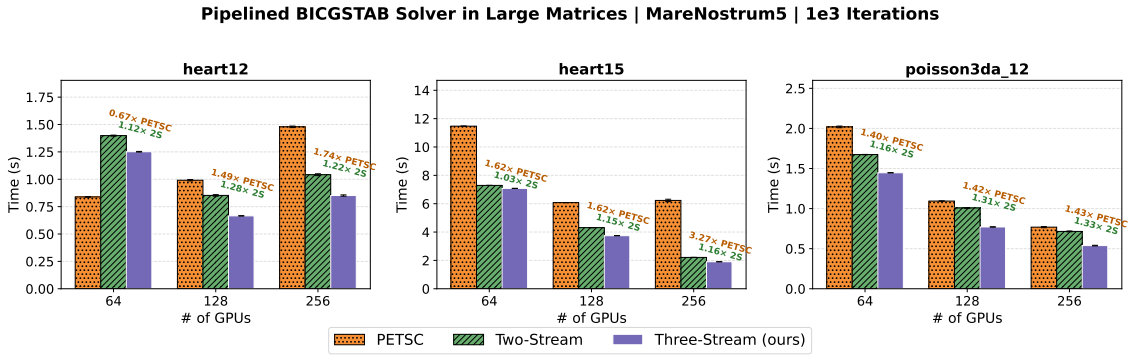


Figure 5.6: Performance of the proposed three-stream design compared with PETSc and the state-of-the-art two-stream implementation of Trotter et al. [Trotter et al., 2025] on the larger matrices on the MareNostrum5 supercomputer. The numbers above the three-stream bars show the speedup over the two-stream implementation and PETSc's Pipe-BICGSTAB implementation.

opportunity for overlap— grows again.

We now repeat this evaluation for the pipelined BiCGSTAB solver. Figure 5.3 reports the SuiteSparse results on MareNostrum 5, and Figure 5.6 shows the results for the three large instances. As before, the bars indicate the solver time for PETSc, the two-stream baseline [Trotter et al., 2025], and our three-stream design.

The qualitative picture matches our Pipe-PCG observations, the three-stream design outperforms the two-stream implementation on every instance and at every GPU count. The margin grows consistently, from 1–5% at 4 GPUs to 17–42% at 64 GPUs. At 64 GPUs, the geometric mean speedup is $1,30\times$ over the two-stream baseline and $2,01\times$ over PETSc. Again, this aligns with our cost model in Chapter 4, as device counts grow, the Allreduce becomes more expensive, which allows the third stream to hide more collective time behind the P2P halo exchange. Pipe-BiCGSTAB is a particularly favorable target for our design. Each iteration issues two reduction phases rather than one, which exposes twice as many opportunities per iteration to overlap the collective with the halo exchange. This advantage is clearly visible in the speedups over PETSc, which are noticeably larger than the corresponding Pipe-PCG numbers on the same machine.

Two instances diverge from this trend. On Geo_1438, the advantage peaks at 16 GPUs ($1,29\times$) before falling to $1,17\times$ at 64 GPUs. On Flan_1565, it peaks at 32 GPUs ($1,35\times$) before dropping to $1,25\times$. In both cases, the solver time at the largest GPU counts is small enough that run-to-run variability becomes comparable to the performance difference between implementations. Furthermore, the massive $4,94\times$ speedup over PETSc on Flan_1565 at 32 GPUs appears to be an outlier caused by temporal environmental instability rather than a structural gain, and we do not draw broader conclusions from it.

Table 5.3 reports the BiCGSTAB speedups over both baselines at 16, 32, and 64 GPUs. Similar to Pipe-PCG, the MPI-based PETSc is consistently the slowest. The gap between PETSc and the stream-based implementations is substantially larger than the gap between the two stream-based designs themselves.

Turning to scaling, we observe the same saturation seen in the Pipe-PCG experiments. Every matrix improves initially, but the scaling curves flatten well before

Table 5.3: Speed-ups of three-stream (ours) over PETSc and two-stream for Pipe-BiCGStab on MareNostrum5 (H100) at 16, 32, and 64 GPUs.

	16 GPUs		**32 GPUs**		**64 GPUs**	
Matrix	**vs. PETSc**	**vs. 2-str.**	**vs. PETSc**	**vs. 2-str.**	**vs. PETSc**	**vs. 2-str.**
Queen_4147	1.47×	1.13×	1.63×	1.33×	1.73×	1.42×
Bump_2911	1.79×	1.24×	2.33×	1.31×	2.65×	1.34×
Flan_1565	1.95×	1.24×	4.94×	1.35×	2.19×	1.25×
audikw_1	1.68×	1.29×	1.88×	1.28×	1.83×	1.32×
Serena	1.96×	1.28×	2.26×	1.20×	2.31×	1.34×
Geo_1438	1.94×	1.29×	2.46×	1.19×	1.57×	1.17×
Geomean	1.79	1.25	2.41	1.27	2.01	1.30

64 GPUs, where all instances except Queen_4147 converge to comparable runtimes. At this point, too little work remains per GPU to justify additional devices, which motivates our analysis of the larger instances.

Figure 5.6 shows the results for the larger instances at 64, 128, and 256 GPUs. Because these problems keep each GPU busy for much longer, the communication fraction is smaller at lower GPU counts. Consequently, the three-stream advantage starts modestly; $1,12\times$, $1,03\times$, and $1,16\times$ on heart12, heart15, and poisson3da_12 at 64 GPUs, respectively. However, the advantage grows on all three instances as the number of GPUs increases, reaching $1,22\times$, $1,16\times$, and $1,33\times$ at 256 GPUs. In short, these larger instances follows a similar trend that the SuiteSparse matrices exhibit before running out of strong scalability. The reason seems to be same, the collective grows more expensive relative to local work, providing more latency for the third stream to hide.

Finally, PETSc is again highly competitive on these large instances. On heart12 at 64 GPUs, PETSc is actually the fastest of the three implementations (making our design $0,67\times$ slower). This highlights the regime where the pipelined formulation offers the least benefit, when the local problem is massive, the collective is

comparatively cheap, and the extra vector operations of the pipelined algorithm are not yet amortized. However, this effect disappears quickly as we scale the problem thinly: at 128 and 256 GPUs, our three-stream design is $1,49\times$ and $1,74\times$ faster than PETSc on the exact same matrix.

5.3 Performance Analysis

So far we have reported end-to-end solver times. In this section we look inside a single solve to explain where the time goes and why the three-stream advantage grows with the number of GPUs.

We profiled two matrices on MareNostrum 5. Queen_4147 is the largest SuiteSparse matrix and scales almost linearly. Flan_1565 is much smaller and reaches its scaling limit early. Together they cover the range of behaviour we observed in Section 5.2. For Queen_4147 we collected a full kernel breakdown from 1 to 64 GPUs, shown in Table 5.4.

5.3.1 How we measure

We time each compute kernel directly with CUDA events. We do not report event-based times for communication that runs on a non-blocking stream. The reason is that such an interval also contains the time the stream spends waiting for its dependencies. It therefore reports far more than the true cost of the operation, and in the three-stream design it can even exceed the total solver time. We only trust the reduction time measured in the two-stream configuration, where the reduction runs on the compute stream and its interval is meaningful. We call this measured value T_R.

For everything else we use a simple subtraction. We add up all measured kernel times and subtract that sum from the total solver time. We call the remainder the *exposed time*. It contains the communication that the design failed to hide, plus synchronisation and load imbalance. This is the quantity our cost model in Chapter 4 aims to predict, so it is the natural target of our analysis.

Table 5.4: Kernel breakdown for Queen_4147 on MareNostrum 5, in seconds, summed over the whole solver loop. $\sum$ is the total of the measured compute kernels. Exposed is the time that remains after subtracting $\sum$ from the total, that is, the communication, synchronisation and imbalance that the design does not hide. A single GPU performs no communication, so its exposed time is only noise.

		Compute kernels								
GPUs	Variant	T_{SD}	T_{SO}	T_{dot}	T_{Axpy}	T_P	$\sum$	Total	Exposed	(%)
1	2-stream	27.904	—	1.254	4.239	0.678	34.076	34.240	0.164	(0.5)
	3-stream	27.898	—	1.249	4.249	0.675	34.071	34.065	0.000	(0.0)
4	2-stream	7.027	0.308	0.429	1.025	0.202	8.991	9.338	0.347	(3.7)
	3-stream	7.022	0.309	0.422	1.028	0.201	8.983	9.179	0.196	(2.1)
8	2-stream	3.535	0.293	0.293	0.492	0.134	4.747	5.196	0.449	(8.6)
	3-stream	3.533	0.286	0.292	0.494	0.131	4.735	4.982	0.247	(5.0)
16	2-stream	1.897	0.273	0.270	0.188	0.086	2.713	3.248	0.534	(16.5)
	3-stream	1.911	0.269	0.275	0.184	0.085	2.724	2.928	0.204	(7.0)
64	2-stream	0.561	0.199	0.225	0.116	0.069	1.170	2.484	1.314	(52.9)
	3-stream	0.554	0.186	0.224	0.114	0.068	1.145	1.918	0.773	(40.3)

5.3.2 Where the compute time goes

The local SpMV scales well, but not perfectly. We start with the local sparse matrix–vector product on the diagonal block, denoted T_{SD} in Chapter 4. From 1 to 4 GPUs it scales almost perfectly and keeps 99% parallel efficiency, dropping from 27.904 to 7.027 seconds. At 64 GPUs it takes 0.561 seconds instead of the ideal 0.436 seconds, so efficiency falls to 78%. Converted into effective memory bandwidth, Queen_4147 reaches 82% of the theoretical peak at 4 GPUs and 65% at 64 GPUs. Flan_1565 falls much faster, from 56% to 9.8%, which simply means that this matrix is too small for a machine of that size.

Some loss is expected even at small scale. Table 2.3 shows that a single H100 reaches about 90% of its theoretical bandwidth on simple streaming kernels. An SpMV only approaches that number under ideal conditions, and many matrices need reordering to improve locality before they can get close as discussed in [Trotter et al., 2023].

The other kernels stop scaling for two different reasons. Table 5.4 lets us tell these reasons separately. The first is growing work. The off-diagonal SpMV T_{SO} improves only from 0.308 to 0.199 seconds between 4 and 64 GPUs, a gain of 1,55× for 16× more GPUs. This is expected, because the number of elements each GPU must receive grows as the matrix is partitioned into more parts, which cancels out the smaller local workload. The AXPY operations behave differently and still have enough work to scale, keeping 55% efficiency over the same range as they drop from 1.025 to 0.116 seconds.

The second reason is kernel-launch overhead. The dot products T_{dot} and the preconditioner T_P run out of work per launch. At 64 GPUs the two dot-product kernels take about 22 μs per iteration, which is roughly 11 μs per launch. That figure sits inside the 5–20 μs launch overhead of the runtime, while perfect scaling from the single-GPU measurement predicts only 2 μs of real work. The preconditioner shows the same pattern, with about 7 μs per iteration against roughly 1 μs of work.

Launch overhead does not depend on the matrix, and neither does the reduction cost T_R. This explains an observation from Section 5.2. At 64 GPUs the smaller

Table 5.5: Measured reduction cost T_R against the compute work available to hide it, for Queen_4147 on MareNostrum 5. We take T_R from the two-stream runs, where the reduction is serialised on the compute stream. The two-stream design can only overlap the diagonal SpMV, and it does not overlap the reduction at all. The three-stream design can also hide the reduction behind the off-diagonal SpMV and the preconditioner, so $M = T_{SD} + T_{SO} + T_P$. The last column is the share of T_R that the third stream removes from the exposed time, computed as $(\text{Exposed}_{\text{2S}} - \text{Exposed}_{\text{3S}})/T_R$.

GPUs	T_R (s)	M_{2S} (s)	M_{3S} (s)	T_R/M_{3S}	Reduction hidden (%)
4	0.156	7.027	7.533	0.02	97
8	0.254	3.535	3.950	0.06	80
16	0.341	1.897	2.265	0.15	97
64	0.844	0.561	0.808	1.04	64

SuiteSparse instances all converge to roughly 1.5 seconds and Queen_4147 to about 2 seconds, because at that scale these matrix-independent overheads set the floor on the running time. Queen_4147 shows it clearly, only 1.17 seconds of its 2.48 second two-stream runtime is compute.

5.3.3 Why the third stream pays off at scale

The cost model of Chapter 4 bases on an interaction between two quantities, the cost of the reduction, T_R, and the amount of compute work available to hide it, M. Table 5.5 shows that relations directly.

Both quantities move in the wrong direction as we add GPUs. The reduction gets more expensive, growing 5,4× from 0.156 seconds at 4 GPUs to 0.844 seconds at 64 GPUs. At the same time the work available to hide it shrinks by roughly an order of magnitude. The ratio T_R/M therefore rises from 0.02 to 1.04.

This crossover is the central result of our analysis. Up to 32 GPUs the reduction is small compared to the compute work, so the extra stream absorbs almost all of it,

the third stream removes between 80% and 97% of T_R from the critical path at 4 to 16 GPUs. At 64 GPUs the reduction has grown until it matches the entire overlap window, since $T_R \approx M$. From this point on the third stream can hide only about 64% of it, and the rest stays exposed no matter how the work is scheduled.

The two-stream design never had this option. Its reduction runs on the compute stream, so T_R is exposed in full at every scale. It costs 1.7% of the two-stream runtime at 4 GPUs but 34% at 64 GPUs. The comparison at 64 GPUs is striking: the exposed reduction alone, 0.844 seconds, is larger than the whole exposed time of the three-stream design, 0.773 seconds. Most of the two-stream overhead therefore comes from how the work is scheduled, not from the speed of the network.

5.3.4 From communication savings to end-to-end speedup

We can now explain the speedups reported in Section 5.2. The three-stream design saves a similar share of communication time at every scale. What changes is how much that share is worth.

At 4 GPUs the saving is already large. Our design cuts the exposed time by 43%, from 0.347 to 0.196 seconds. But exposed time is only 3.7% of the two-stream solve at this scale, and 2.1% of ours, so the end-to-end gain is just 1,02×. This matches the small speedups we see at 4 GPUs in Figure 5.1.

As we add more GPUs, the same saving applies to a much larger share of the runtime. The exposed fraction of the two-stream solve grows from 3.7% at 4 GPUs to 8.6% at 8 GPUs, 16.5% at 16 GPUs and 52.9% at 64 GPUs. At that point communication and synchronisation dominate the solver runtime, not computation. Our design holds the exposed fraction at 40.3%. It cuts the exposed time by 41%, from 1.314 to 0.773 seconds, and this turns into a 1,29× end-to-end speedup. The intermediate points follow the same path, with gains of 1,04× at 8 GPUs and 1,11× at 16 GPUs.

This progression is exactly what the cost model predicts. A large communication saving stays invisible while communication is a small part of the runtime, and becomes decisive once it starts the dominate the overall runtime. The same reason-

ing explains the two extremes we saw earlier. Under strong scaling the advantage widens, because T_R grows while M shrinks. On the very large compute-heavy instances the advantage may narrow, because local compute load gets larger, thus M stays large enough that even a two-stream approach leaves little of the reduction exposed.

Chapter 6

RELATED WORK

Prior work on communication overlap in conjugate gradient (CG) methods typically treats collective communication and point-to-point (P2P) halo exchanges as separate concerns. Existing approaches overlap only one of these communication classes with computation, leaving the other serialized on the critical path. Our thesis bridges this gap by scheduling collective reductions, halo exchanges, and local computations as three concurrent streams. We divide the related work into four areas: pipelined Krylov methods, halo overlap in iterative solvers, GPU-side communication strategies, and the joint scheduling of communication classes.

6.1 Pipelined and Communication-Avoiding Krylov Methods

Pipelined CG methods reduce the global synchronization overhead that limits Krylov solvers at scale. Ghysels and Vanroose [Ghysels and Vanroose, 2014] reformulated preconditioned CG to issue a single non-blocking reduction per iteration, hiding its latency behind the SpMV and preconditioner. Subsequent works extended this with deeper pipelines to hide reductions behind multiple SpMVs [Cornelis et al., 2018], reduced the frequency of reductions [Tiwari and Vadhiyar, 2020], or incorporated variable preconditioners [Sanan et al., 2016]. On CPU clusters, these methods rely heavily on runtime asynchronous progress [Denis et al., 2022].

However, this entire line of work remains reduction-centric. It targets the global dot product latency while treating the P2P halo exchange as part of the local computation. We build on this pipelined formulation but depart from it by promoting the halo exchange to an independent communication stream. This allows us to hide the reduction behind both computation and the halo exchange.

6.2 Halo and P2P Overlap in Iterative Solvers and Multigrid

A separate line of research overlaps the P2P halo communication of iterative solvers and multigrid methods with computation. These works aim to hide nearest-neighbor exchanges behind local SpMV or preconditioner work. Researchers have explored this using OpenMP dynamic scheduling [Nakajima et al., 2022, Nakajima, 2025], asynchronous progress on manycore CPUs [Horikoshi et al., 2022], and load balancing across CPU-GPU systems for unstructured meshes [Langguth et al., 2015]. Infrastructure libraries like Tausch [Spies et al., 2022] further lower the engineering cost of this overlap.

These works primarily address the halo exchange in isolation. They do not consider jointly scheduling the halo exchange and the collective reduction. Our design treats both communication classes as concurrent, optimizing both rather than just one.

6.3 GPU-Side Communication Strategies for CG

On GPU systems, recent work explores communication overlap from an implementation perspective. Researchers have reduced synchronizations in multi-GPU SpMVs [Bernaschi et al., 2025], used persistent kernels for device-initiated communication [Ismayilov et al., 2023], and optimized specific SpMV and preconditioner kernels [Ma et al., 2025]. Trotter et al. [Trotter et al., 2025] compared CPU- and GPU-initiated strategies across GPU-aware MPI, NCCL/RCCL, and NVSHMEM, identifying an NCCL-based design that outperforms GPU-aware MPI. We use this state-of-the-art design as our two-stream baseline in Chapter 5.

Other works provide portable high-level communication libraries [Sağbili et al., 2025] and tuning recommendations for modern supercomputers [De Sensi et al., 2024], which we adopt. Despite these advances, none of these approaches schedule reductions, halo exchanges, and computation as three concurrent streams. They optimize individual primitives or shift initiation to the GPU, but still overlap at most one communication class with computation.

6.4 Joint Scheduling of Communication Classes

The closest prior work is by Selvitopi et al. [Selvitopi et al., 2015], who embed the irregular P2P SpMV messages into a single collective communication phase. While this mixes both classes, it fuses them into one collective rather than running them concurrently. Other related ideas appear in user-level thread frameworks [Lu et al., 2015] and distributed Mixture-of-Experts training [Li et al., 2023]. For example, Lina overlaps all-to-all and Allreduce collectives on separate streams, but notes this can degrade performance due to network contention.

Our approach avoids this bottleneck. By overlapping a collective reduction with a P2P halo exchange, the localized P2P messages place far less pressure on network bandwidth than two competing global collectives. This complementary resource usage makes our concurrent three-stream schedule profitable. We achieve the true three-way overlap that prior joint-scheduling work misses, and we ground our design choices in the analytical cost model presented in Chapter 4.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Inter-GPU communication overhead is a primary obstacle to scaling sparse iterative solvers. In a distributed pipelined preconditioned conjugate gradient (Pipe-PCG) solve, this overhead takes two forms, latency-bound collective reductions (from dot products) and bandwidth-bound point-to-point (P2P) halo exchanges (from distributed SpMV). The traditional two-stream approach overlaps only one communication class with computation. It serializes the other, leaving a significant latency-hiding opportunity unexploited.

This thesis introduces a three-stream technique to overlap both communication types with computation and with each other. Because collective reductions are latency-bound and halo exchanges are bandwidth-bound, they stress complementary system resources and run concurrently without contention. We assign collective reduction, halo exchange, and SpMV to three separate streams using modern libraries like NCCL. To achieve genuine concurrency, we utilize separate communicators and configuration options to prevent the streams from blocking one another.

We developed an analytical cost model to evaluate this design. It predicts that three-stream benefits grow as the Allreduce cost increases—specifically at higher GPU counts and in strong-scaling regimes. Our experiments on the MareNostrum 5 and Leonardo supercomputers confirmed this prediction. On SuiteSparse test matrices, our three-stream design consistently outperformed the two-stream baseline. The performance gap widened from roughly 2–6% at 4 GPUs to 18–47% at 64 GPUs.

Overall, we achieved up to a 33% speedup on 256 GPUs, and up to a 47% speedup on 64 GPUs on Leonardo and MareNostrum5 supercomputers. At 64 GPUs, geometric-mean speedups over the two-stream method were 1,33× (MareNostrum

5) and 1,23× (Leonardo) in Pipe-PCG solver and 1,30x (MareNostrum 5) in Pipe-BICGSTAB solver. Compared to PETSc, which uses GPU-aware MPI, our speedups reached 1,38× (MareNostrum 5) and 1,72× (Leonardo) in Pipe-PCG solver and 2,41x (MareNostrum 5) in Pipe-BICGSTAB solver. Since the theoretical maximum improvement is 100%, these 47% gains are highly significant. Ultimately, these results demonstrate that our three-stream technique successfully mitigates communication bottlenecks at scale.

7.2 Limitations

Several limitations bound the scope of our results. First, we limit our implementation and evaluation to NVIDIA GPUs and the NCCL communication library. Although the three-stream concept is vendor-independent, we have not yet evaluated it on other hardware or communication stacks. Second, our Pipe-PCG solver uses a Jacobi preconditioner. This choice keeps the preconditioner application inexpensive and local, but it does not reflect the more elaborate preconditioners often used in real-life applications. The interaction between our three-stream approach and a heavier, communication-bearing preconditioner remains unexplored.

Third, our achieved concurrency relies on advisory contention hints and configuration options. We ask the runtime to let the two communication streams progress independently, but it does not strictly guarantee this isolation. Therefore, the actual degree of overlap depends heavily on the underlying library and hardware behavior. Finally, system-level job allocation policies restricted the maximum cluster size we could evaluate. As a result, we infer the extreme-scale behavior of our design from scaling trends rather than direct measurements.

7.3 Future Work

Several future directions follow naturally from this work. First, we plan to extend the three-stream technique to support AMD GPU architectures. We will use the ROCm Collective Communication Library (RCCL), which mirrors the stream-based communication model of NCCL. This port will allow us to assess whether our overlap

benefits hold across a different hardware interconnect and software stack.

Second, we aim to apply this technique beyond Krylov subspace solvers. The three-stream principle works for any large-scale distributed-memory application combining substantial local data movement with global coordination. For example, many optimization and simulation codes interleave neighbor exchanges with periodic convergence checks or global best-solution updates. Demonstrating triple overlap in these workloads would prove the broad generality of our approach.

Third, we plan to explore deeper pipelines. Variants of pipelined CG and pipelined BiCGSTAB expose more independent work per iteration. This extra computational work provides additional slack to hide both classes of communication concurrently. At extreme scales, this could significantly amplify the performance benefits of our approach.

Pursuing these directions will elevate the three-stream technique. It will evolve from a specialized optimization for Krylov solvers into a general, highly effective strategy for hiding mixed collective and P2P communication on GPU supercomputers.

BIBLIOGRAPHY

[Banchelli et al., 2025] Banchelli, F., Garcia-Gasulla, M., Mantovani, F., Vinyals, J., Pocurull, J., Vicente, D., Eguzkitza, B., Galeazzo, F. C., Acosta, M. C., and Girona, S. (2025). Introducing MareNostrum5: A European pre-exascale energy-efficient system designed to serve a broad spectrum of scientific workloads. *Future Generation Computer Systems*, page 108125.

[Bernaschi et al., 2025] Bernaschi, M., Carrozzo, M., Celestini, A., Piperno, G., and D'Ambra, P. (2025). Communication-reduced conjugate gradient variants for GPU-accelerated clusters. In *2025 33rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 178–186.

[Community, 2026] Community, T. O. (2026). 11.3.5. CUDA — Open MPI main documentation. `https://docs.open-mpi.org/en/main/tuning-apps/accelerators/cuda.html`.

[Cools, 2019] Cools, S. (2019). Analyzing and improving maximal attainable accuracy in the communication hiding pipelined BiCGStab method. *Parallel Computing*, 86:16–35.

[Cools and Vanroose, 2017] Cools, S. and Vanroose, W. (2017). The communication-hiding pipelined BiCGStab method for the parallel solution of large unsymmetric linear systems. *Parallel Computing*, 65:1–20.

[Cornelis et al., 2018] Cornelis, J., Cools, S., and Vanroose, W. (2018). The communication-hiding conjugate gradient method with deep pipelines. *CoRR*, abs/1801.04728.

[Davis and Hu, 2011] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1).

[De Sensi et al., 2024] De Sensi, D., Pichetti, L., Vella, F., De Matteis, T., Ren, Z., Fusco, L., Turisini, M., Cesarini, D., Lust, K., Trivedi, A., Roweth, D., Spiga, F., Di Girolamo, S., and Hoefler, T. (2024). Exploring GPU-to-GPU communication: Insights into supercomputer interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press.

[Denis et al., 2022] Denis, A., Jaeger, J., Jeannot, E., and Reynier, F. (2022). A methodology for assessing computation/communication overlap of mpi non-blocking collectives. *Concurrency and Computation: Practice and Experience*, 34(22):e7168.

[Ghysels and Vanroose, 2014] Ghysels, P. and Vanroose, W. (2014). Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Comput.*, 40(7):224–238.

[Hestenes and Stiefel, 1952] Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–435.

[Horikoshi et al., 2022] Horikoshi, M., Gerofi, B., Ishikawa, Y., and Nakajima, K. (2022). Exploring communication-computation overlap in parallel iterative solvers on manycore cpus using asynchronous progress control. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops*, HPCAsia '22 Workshops, page 29–39, New York, NY, USA. Association for Computing Machinery.

[Hu et al., 2026] Hu, Z., Shen, S., Bonato, T., Jeaugey, S., Alexander, C., Spada, E., Dinan, J., Hammond, J., and Hoefler, T. (2026). Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms.

[Ismayilov et al., 2023] Ismayilov, I., Baydamirli, J., Sağbili, D., Wahib, M., and Unat, D. (2023). Multi-gpu communication schemes for iterative solvers: When

cpus are not in charge. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 192–202, New York, NY, USA. Association for Computing Machinery.

[Langguth et al., 2015] Langguth, J., Sourouri, M., Lines, G. T., Baden, S. B., and Cai, X. (2015). Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes . *IEEE Micro*, 35(04):6–15.

[Li et al., 2023] Li, J., Jiang, Y., Zhu, Y., Wang, C., and Xu, H. (2023). Accelerating distributed MoE training and inference with Lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA. USENIX Association.

[Lu et al., 2015] Lu, H., Seo, S., and Balaji, P. (2015). Mpi+ult: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454.

[Ma et al., 2025] Ma, T., Gou, Z., Xu, N., and Sun, S. (2025). Efficient multi-GPU implementations of preconditioned conjugate gradient method. *Advances in Engineering Software*, 207:103936.

[Martinez-Navarro et al., 2019] Martinez-Navarro, H., Rodriguez, B., Bueno-Orovio, A., and Minchole, A. (2019). Repository for modelling acute myocardial ischemia: simulation scripts and torso-heart mesh.

[Merrill and Garland, 2016] Merrill, D. and Garland, M. (2016). Merge-based parallel sparse matrix-vector multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689.

[Mills et al., 2024] Mills, R. T., Adams, M., Balay, S., Brown, J., Faibussowitsch, J., Isaac, T., Knepley, M., Munson, T., Suh, H., Zampini, S., Zhang, H., and Zhang, J. (2024). Petsc/tao developments for gpu-based early exascale systems.

[Nakajima, 2025] Nakajima, K. (2025). Optimization of communication-computation overlapping in parallel multigrid methods by process/thread allocation. *Japan Journal of Industrial and Applied Mathematics*, 42(3):1029–1062.

[Nakajima et al., 2022] Nakajima, K., Gerofi, B., Horikoshi, M., and Ishikawa, Y. (2022). Communication-computation overlapping for preconditioned parallel iterative solvers with dynamic loop scheduling. *International Conference on High Performance Computing in Asia-Pacific Region Workshops*.

[NVIDIA Corporation, 2022] NVIDIA Corporation (2022). *cuSPARSE Library*. NVIDIA Corporation. [Online].

[NVIDIA Corporation, 2025] NVIDIA Corporation (2025). *cuBLAS Library*. NVIDIA Corporation, release 12.8 edition. [Online].

[NVIDIA Corporation, 2026a] NVIDIA Corporation (2026a). HCOLL — NVIDIA docs. `https://docs.nvidia.com/networking/display/hpcxv2251/hcoll`. [Online; accessed 2026-01-12].

[NVIDIA Corporation, 2026b] NVIDIA Corporation (2026b). NVIDIA collective communications library (NCCL). `https://developer.nvidia.com/nccl`. [Online; accessed April 2026].

[Ruhela et al., 2018] Ruhela, A., Subramoni, H., Chakraborty, S., Bayatpour, M., Kousha, P., and Panda, D. K. (2018). Efficient asynchronous communication progress for mpi without dedicated resources. In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI '18, New York, NY, USA. Association for Computing Machinery.

[Sanan et al., 2016] Sanan, P., Schnepp, S., and May, D. (2016). Pipelined, flexible Krylov subspace methods. *SIAM Journal on Scientific Computing*, 38(5):C441–C470.

[Sağbili et al., 2025] Sağbili, D., Ekmekçibaşı, S., Ibrahim, K. Z., Nguyen, T., and Unat, D. (2025). Uniconn: A uniform high-level communication library for portable multi-gpu programming. In *2025 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12.

[Selvitopi et al., 2015] Selvitopi, R. O., Ozdal, M. M., and Aykanat, C. (2015). A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):632–645.

[Spies et al., 2022] Spies, L., Bienz, A., Moulton, D., Olson, L., and Reisner, A. (2022). Tausch: A halo exchange library for large heterogeneous computing systems using mpi, opencl, and cuda. *Parallel Comput.*, 114(C).

[Tiwari and Vadhiyar, 2020] Tiwari, M. and Vadhiyar, S. (2020). Pipelined preconditioned conjugate gradient methods for distributed memory systems. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 151–160.

[Trotter et al., 2023] Trotter, J. D., Ekmekçibaşı, S., Langguth, J., Torun, T., Düzakın, E., Ilic, A., and Unat, D. (2023). Bringing order to sparsity: A sparse matrix reordering study on multicore cpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA. Association for Computing Machinery.

[Trotter et al., 2025] Trotter, J. D., Ekmekçibaşı, S., Sağbili, D., Langguth, J., Cai, X., and Unat, D. (2025). Cpu- and gpu-initiated communication strategies for conjugate gradient methods on large gpu clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and*

Analysis, SC '25, page 298–315, New York, NY, USA. Association for Computing Machinery.

[Turisini et al., 2023] Turisini, M., Amati, G., and Cestari, M. (2023). Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications.