juliacon

XK.jl: Composable and Portable Multi-GPU BLAS in Julia

Romain PEREIRA[1], Alexis MONTOISON[1], Michel SCHANEN[1], and Swann PERARNAU[1]

[1]Argonne National Laboratory, Lemont, IL 60439, USA

ABSTRACT

This paper introduces XK.BLAS: the BLAS module of the Julia package XK.jl. This module provides BLAS APIs that are portable across all three major GPU vendors (AMD, Intel, NVIDIA) for multi-device architectures. XK.BLAS is built on top of the XKRT tasking runtime systems and the XKBlas library. An XKBlas program is a sequence of task-generating routines with no explicit handling of memory (such as allocation and data movement) or scheduling decisions (such as target device selection). Memory is instead lazily allocated, migrated, and evicted, while tasks are automatically distributed across available devices. Tasks are composable, so that fine-grained dependencies are inferred from the memory regions they access, allowing concurrent data movement and kernel execution across different routines when the dataflow permits. We evaluate primitives of XK.BLAS and its use as a multi-GPU backend for the package Krylov.jl. We show up to 2.8× speedup when scaling from 1 to 4 H100 GPUs. More importantly, XK.BLAS enables solving problems that would not otherwise fit on a single GPU, with no code changes at all.

Keywords

Julia, BLAS, Macro-dataflow

1. Introduction

The Basic Linear Algebra Subprograms (BLAS [7]) specify routines for scientific computing. Original implementation (Netlib, OpenBLAS) target CPUs. GPU companies (AMD, Intel, NVIDIA) provide high-performance implementation (rocBLAS [24], OneMKL [13], cuBLAS [22]) for GPUs. These libraries are fundamental building blocks for developing scientific software such as iterative solvers [20, 2]. However, despite the advent of multi-GPU servers, vendors' implementation only targets single-GPU: programmers are responsible for manually parallelizing execution across multiple devices (i.e., moving memory and synchronizing kernel executions).

This motivated the design of new multi-GPU BLAS implementation [25, 18, 23, 9, 11, 1, 16, 4] that automates parallel execution from a sequential program that preserves historical BLAS APIs. They are all implemented in C/C++ and tile routines [12] to distribute tasks to multiple devices. Among these libraries, XKBlas [18] uniquely provides highly composable tasking interfaces with relaxed synchronizations between routines [1]. It has shown great success in accelerating the sparse solver MUMPS [3] to multiple GPUs with only a few lines of code changes ($\simeq 10-20$ LoC) with great performance portability thanks to its underlying tasking model.

The Julia linear algebra ecosystem currently lacks similar libraries to automatically move memory and schedule BLAS routines on multi-device architectures. Current practices involve manually moving memory and distributing work via vendor-specific libraries (AMDGPU.jl, CUDA.jl, oneAPI.jl), and dispatching linear algebra primitives depending on input data types (`ROCArray`, `CuArray`, `oneArray`). A promising ongoing effort is cuNumeric.jl [19], inspired by cuPyNumeric and built on top of Legion [8]. However, it is currently restricted to NVIDIA GPUs and remains at an early stage of development.

This paper fills that gap in the Julia ecosystem by introducing the BLAS module of XK.jl (XK.BLAS): a package built on top of XKRT and XKBlas automating data motions and execution of BLAS across multi-device architectures. Our contributions are the following:

- The introduction of XK.BLAS provides composable and portable multi-GPU BLAS in Julia.
- Extensions to XKBlas to support BLAS 1 and 2 routines, that can be used as drop-in replacements in CPU programs (i.e., automatic multi-GPU parallelization with no code change at all).
- The implementation of a new backend for Krylov.jl [20], a collection of Krylov solver, using XK.BLAS to provide performant and portable parallelization on most of its solvers across multi-GPU architectures.

The paper is organized as follows. Section 2 introduces the background of XK.BLAS. Section 3 presents its implementation and the extensions enabling improved support for Krylov solvers. Section 4 evaluates XK.BLAS primitives and their integration as a multi-GPU backend for Krylov.jl. Section 5 concludes the paper.

2. Background

XK.jl is a Julia framework for composable and portable multi-GPU linear algebra. Its core module, XK.BLAS enables sequential CPU-like code to run efficiently across multiple GPUs by automatically managing memory, scheduling tasks, and overlapping computation and communication. We first introduce Krylov.jl as a motivating application, and discuss how its reliance on BLAS routines motivates the extensions and multi-device capabilities of XK.BLAS.

2.1 Krylov.jl

The package Krylov.jl implements over 35 Krylov methods for solving a variety of problems, including square linear systems, least-squares problems, least-norm problems, and structured 2×2 block systems. It is the primary motivation for the BLAS module

[1] https://gitlab.inria.fr/xkblas/dev/tree/v2.0

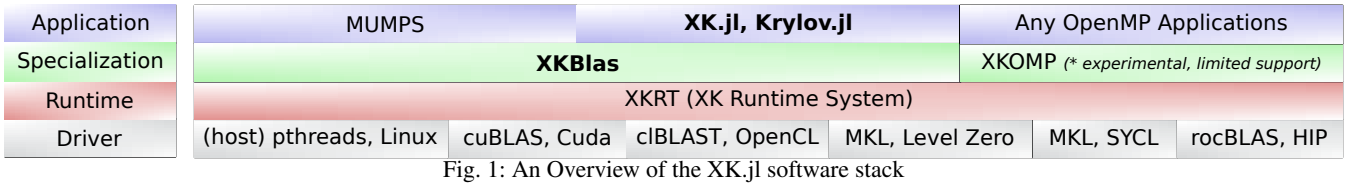

Fig. 1: An Overview of the XK.jl software stack

Code 1: A few lines of code from Krylov.jl implementation of the conjugate gradient method.

```
1 function cg(A, b; kwargs...)
2     # [...]
3     kmul!(Ap, A, p)        # Ap  := A * p
4     pAp = kdotr(n, p, Ap) # pAp := real(p'Ap)
5     # [...]
6     α = γ / pAp
7     # [...]
8 end
```

of XK.jl. Krylov.jl specifies a set of routines that need to be implemented to execute its solvers agnostically across devices. On the CPU, Krylov.jl dispatches these routines through Libblastrampoline (LBT), allowing seamless switching between different BLAS backends such as Intel MKL, AOCL, Apple Accelerate, or BLIS. On GPUs, multiple dispatch is used based on the type of the GPU vector (like `CuVector` or `ROCVector`) to target the appropriate GPU BLAS implementation.

Solvers are implemented as if they were executing on a single CPU, with no considerations on memory locality, nor expressing any parallelism explicitly. We illustrate it on code 1 with a few lines of code extracted from its implementation of the conjugate gradient method. Line 3 runs the operator `kmul!` between the variables A and p and stores the result in the variable Ap. For instance, if Ap, A, and p respectively are `CuVector`, `CuMatrix`, and `CuVector`, then Krylov.jl dispatches to the `gemv` implementation of cuBLAS assuming the memory is already present and coherent on the device. The executing thread waits for the operator completion, and then on line 2, it computes the dot product and stores the real part of the results to the variable `pAp` in the host memory. Line 4, it computes $\alpha = \gamma / pAp$ on the host.

Currently, GPU support relies on GPUArrays.jl and its AMDGPU.jl, CUDA.jl, and oneAPI.jl specializations. Programmers must explicitly and synchronously move memory from the host to the device before solver execution by declaring `ROCArray`, `CuArray`, or `oneArray` data structures. The Krylov.jl backend then dispatches to the respective vendor-specific single-GPU BLAS libraries using Julia's multiple dispatch feature. The primary motivation of this work is to provide multi-GPU support for Krylov.jl solvers without modifying its existing 12,000 lines of code, which implement the Krylov solvers. All methods heavily rely on BLAS 1 and 2 routines, while two of them, the block Krylov methods block-MINRES and block-GMRES, also use LAPACK and BLAS 3 routines. These remain as future work.

2.2 XKBlas, XKRT

The XKBlas library [18] accelerates BLAS 3 in the MUMPS linear solver [3] on multi-GPU architectures. This work shares the same original motivations: accelerate a sequential host program to multiple devices. It was originally built on top of the Kaapi task-

Code 2: Pseudo-code of an XKBlas program. The `trsm_async` and `gemm_async` spawn tasks executing the routine on submatrices, that synchronizes depending on sequential data dependencies. The `memory_host_coherent_async` spawns a task that writes-back the passed memory region (i.e., the matrix C) to its original host replica.

```
1  # A, B and C are matrices on the host memory
2  # Routines are executed on available devices
3
4  # trsm reads A, B and writes B
5  trsm_async(A, B,    tile_size=n/2);
6
7  # gemm reads A, B, C and writes C
8  gemm_async(A, B, C, tile_size=n/2);
9
10  # ask for host write-back
11 memory_host_coherent_async(C);
12
13 # wait for the completion of previous tasks
14 sync();
15
16 # At this point of the execution, the matrix
17 # - C was written-back to the host memory.
18 # - B was not.
```

ing runtime system [17] and had been recently (2024-2025) rewritten on top of the XKRT runtime system [2]—a fork of Kaapi that supports byte-granular data dependencies and motion for macro-dataflow tasks ala Legion [8]. We summarize the software stack in Figure 1. The key aspects of XKBlas are:

- (1) Automated memory management. XKBlas APIs use host memory pointers and lazily replicates on the executing devices' memories. XKBlas book-keeps memory coherence over routine invocations, so that the memory is assumed *coherent* (aka. valid) only on the device that lastly wrote it.
- (2) Multi-GPU support. XKBlas internally tiles and distributes BLAS routines to available devices.
- (3) Asynchrony. XKBlas overlaps GPU kernel execution with data motion implicitly while ensuring a partial and correct parallel order of execution following the sequential program data dependencies that are well-defined by the BLAS specification: for instance, `gemm(A, B, C)` reads A, B, and C, and writes C.

An XKBlas task includes a kernel and a declaration of its memory accesses (read, write, and the host virtual memory region) so that a correct parallel order of execution can be deduced from data hazards (read-after-write, etc., on intersecting regions of memory) by its underlying runtime system. An XKBlas program is a sequence of CPU-like routines spawning tasks that compose automatically to overlap independent data motion and computation—as illustrated on Code 2. The routines (`gemm_async`) lines 3 and 4 spawn tasks

[2] https://github.com/rpereira-dev/xkrt

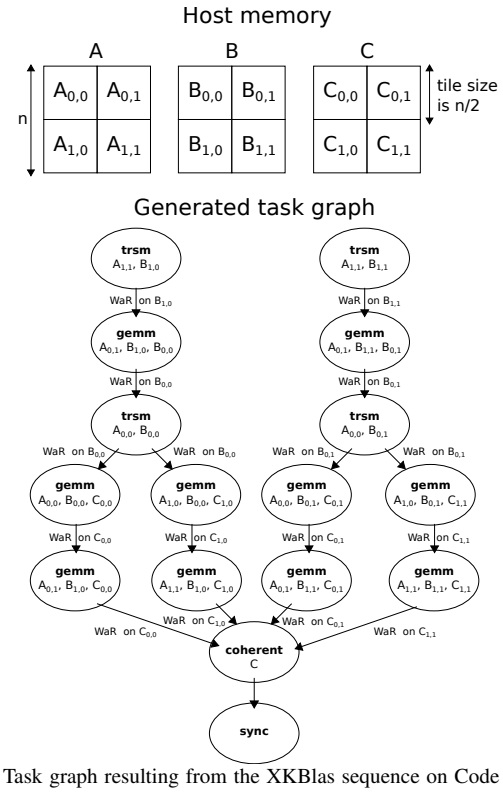

Fig. 2: Task graph resulting from the XKBlas sequence on Code 2. Tasks and their associated memory are distributed to available GPUs in parallel, and concurrently to the graph creation. Minimal data motions are ensured from conflicting memory accesses by the XKRT internal coherence protocol.

to be executed and return before their completion. The `sync` line 5 waits for the completion of all tasks previously spawned. The `memory_coherent_async` line 11 spawns an empty task on the host to write-back the matrix D so the host memory is coherent by the completion of the `sync` line 12. We illustrate the resulting task dependency graph in Figure 2: independent tasks can execute in parallel on different GPUs.

XKBlas originally only supported BLAS 3 routines that represented the largest potential for GPU acceleration in MUMPS. This paper extends it with support for the BLAS 1 and 2 routines to provide a multi-GPU backend to Krylov.jl.

3. XK.BLAS: multi-devices BLAS in Julia

XK.BLAS provides the core multi-device BLAS functionality of XK.jl. In this section, we present its architecture and runtime (XKRT), the available execution model flavors, and our extensions to support BLAS 1 and 2 routines. We also show how XK.BLAS can serve as a drop-in multi-GPU backend for Krylov.jl, enabling efficient execution without modifying its original CPU-based source code.

3.1 Binding to a C/C++ library

XKBlas and XKRT are C++ libraries that expose a C API. XK.jl automatically generates bindings to these C APIs using Foreign Function Interfaces (FFIs) generated via Clang.jl. On top of these bindings, we manually implement wrappers that provide idiomatic Julia interfaces, allowing seamless integration within the Julia linear algebra programming environment, as illustrated in Code 3. For example, XK.BLAS dispatches routines to the corresponding XKBlas implementation based on parameter precision (`Float32` maps to single, `Float64` to double, etc.). Additionally, its BLAS routines and the `memory_coherent_async` API support native Julia linear algebra types such as `AbstractVector` and `AbstractMatrix`, as shown on line 19.

Code 3: An XK.BLAS program example.

```
1  using LinearAlgebra, Random, XK
2
3  # Run parameters
4  const T = Float32
5  m = n = k = 16_384
6  lda, ldb, ldc = m, k, m
7  alpha = T(1.0)
8  beta = T(0.0)
9  transA = transB = XK.BLAS.NO_TRANS
10
11 # Create host matrices
12 A = Matrix{T}(undef, m, k)
13 B = Matrix{T}(undef, k, n)
14 C = Matrix{T}(undef, m, n)
15
16 # Execution on all available devices - 3 flavors:
17
18 # (F1) Blocking, and automatic write-back
19 XK.BLAS.gemm(
20     transA, transB,
21     alpha, A, B,
22     beta,  C
23 )
24
25 # (F2) Blocking, no automatic write-back
26 XK.BLAS.gemm_sync(
27     transA, transB,
28     alpha, A, B,
29     beta,  C
30 )
31
32 # (F3) Non-blocking, no automatic write-back
33 XK.BLAS.gemm_async(
34     transA, transB,
35     alpha, A, B,
36     beta,  C
37 )
38 XK.BLAS.sync()
```

3.2 Execution model flavors

Additionally, XK.BLAS provides three execution model flavors. The input objects always reside in host memory, but each flavor defines how synchronization and write-back are handled:

- (F1) Drop-in (line 19). The calling thread blocks until all tasks complete, and every byte written on any device is automatically written back to the host. In the code example, the executing threads wait for the `gemm` tasks to finish, and the matrix `C` is implicitly updated on the host.
- (F2) Synchronous (line 25). The calling thread blocks until all tasks complete, but there is no automatic write-back. Programmers can explicitly request write-back via `XK.BLAS.memory_host_coherent_async`. In the code example, the threads wait for the `gemm` tasks, but parts of `C` remain on the devices until explicitly synchronized.
- (F3) Asynchronous (line 33). The calling thread only spawns tasks and returns immediately, without waiting for their

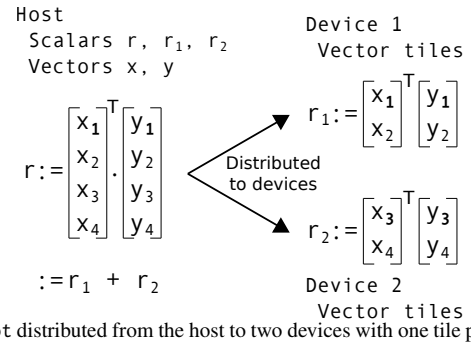

Fig. 3: `dot` distributed from the host to two devices with one tile per device.

completion. Programmers can enforce completion by calling `XK.BLAS.sync()`. As in (F2), there is no automatic write-back. In the example, the tasks are spawned, the function returns, and completion is enforced later via `sync()`.

Originally, XKBlas only supported flavor (F3). We extended it with (F1) and (F2) to serve as drop-in replacements for CPU BLAS programs, requiring no changes to existing code.

3.3 Support for BLAS 1 and 2

The Krylov.jl library relies extensively on BLAS 1 and 2 routines (`axpby`, `dot`, `nrm2`) and matrix vector operations (`gemv`, `spmv`). These routines were not originally supported in XKBlas. We therefore implemented them, and present our approach in this section.
As with the original BLAS 3 routines, BLAS 1 and 2 operations are tiled into tasks and distributed across multiple devices. Each task eventually launches a vendor-specific kernel (cuBLAS, hipBLAS, or oneMKL) on a subvector or submatrix. XKBlas tiles dense matrices and vectors contiguously using a fixed tile size, improving memory locality on devices when chaining multiple kernels. Figures 3 and 4 illustrate our multi-device implementations of `dot` and `spmv`.
Currently, `spmv` only supports the Compressed Sparse Row (CSR) format. Given a sparse matrix of size (m, n), our implementation preprocesses the matrix in four steps:

- $\mathcal{S}_1$–Tile the matrix into blocks of size $(m_t \leq m, n)$ and assign each block to a device,
- $\mathcal{S}_2$–Execute a custom kernel to adjust row indices for each tile so they are valid on the local device memory,
- $\mathcal{S}_3$–Compute the minimum and maximum column indices containing non-zero elements for each tile.
- $\mathcal{S}_4$–Run vendor specific preprocessing routines (`cusparseSpMV_preprocess`, etc.).

The step $\mathcal{S}_3$ is optional. If $\mathcal{S}_3$ is disabled, the entire vector x will be moved on the executing device. For instance on Figure 4, all x_1, x_2, x_3 and x_4 will be copied to the device 2 even though only x_3 and x_4 are needed, since the matrix columns 1 and 2 are only zeroes. However, if $\mathcal{S}_3$ is enabled, the runtime will only copy the smallest sub-vector that includes all lines with non-zeroes columns in the matrix—e.g., on the figure, only x_3 and x_4 are copied—which reduces data movements. Such technique is used by the HPCG benchmark that uses a 7-diagonal matrix distributed with MPI [15]. All steps are performed only the first time the matrix is encountered, and the results are cached for later reuse. Programmers can explicitly invalidate these caches if the matrix values change.

3.4 A Multi-GPU Backend for Krylov.jl

XK.jl introduces several new Julia types, namely `XKVector`, `XKMatrix`, and `XKSparseMatrixCSR`, which act as wrappers around specialized data structures while preserving existing CPU and GPU support through multiple dispatch. These types enable the integration of XK.jl with Krylov.jl without modifying its original programming model.
The XK.BLAS module overloads the default Krylov.jl backend by providing implementations based on the flavor (F2) API. This flavor ensures minimal data movement and remains fully compatible with Krylov.jl's execution semantics (see Section 2.1).
For instance, adopting flavor (F3) would result in an incorrect execution order in Code 1. Specifically, the computation of $\alpha = \gamma/pAp$ (line 6) could be executed before the completion of the `kdotr` operation (line 4). Guaranteeing correct behavior in this case would require modifying Krylov.jl to introduce explicit synchronization points, which we intentionally avoid.
Conversely, flavor (F1) preserves correctness but incurs significant overhead due to frequent host-to-device (H2D) and device-to-host (D2H) memory transfers between solver iterations. By leveraging flavor (F2), XK.BLAS relies on XKRT's internal memory coherence mechanisms to enforce correct execution order while minimizing data movement.

4. Evaluation

We evaluate the performance of XK.BLAS using Julia v1.11.8 on the experimental environments shown in Table 1. First, we evaluate a few of its routines to ensure proper binding between Julia and the underlying C/C++ runtime system, and scalability against the number of GPUs. Then, we evaluate XK.BLAS as a multi-GPU backend for Krylov.jl.

Table 1. : Hardware and software stack used in the evaluations

Name	Hardware (CPU)	Hardware (GPU)	Software
AMD-JLSE	2x AMD EPYC 7713 @2.0GHz	8x MI250X	rocm 6.3.2, LLVM 19.1.0
NV-JLSE	2x Intel(R) Xeon(R) Platinum 8468 @3.8GHz	4x H100	nvhpc 24.1, LLVM 21.x
Aurora	2x Intel Xeon CPU Max Series (SPR) @2.0 GHz	6x PVC	oneapi 24.347.0

4.1 Routines Benchmarking

All experiments of this section report averages and standard deviations over 10 instances of execution. We report results for a routine of each BLAS level on different GPU vendors.

4.1.1 BLAS 1 - axpy. Figure 5 reports performance on an `axpy` scaling from 1 to 8 GPUs, including the initial H2D and final D2H transfers. On AMD-JLSE, GPUs are connected in pairs to the PCIe slots (i.e., 4 PCIe links for 8 GPUs). Due to the low arithmetical intensity of an `axpy`, the performance scaling is mostly related to memory motions: the observed gain on multiple GPUs is due to an increased aggregated bandwidth by using each PCIe link concurrently, for the initial H2D and final D2H transfers. This also explains the lack of scalability from 1 to 2 GPUs that share the same PCIe.

4.1.2 BLAS 2 - spmv. Figure 6 reports performance on an `spmv` scaling from 1 to 4 GPUs on NV-JLSE. Timings only include the computation (i.e., not the preprocessing nor the initial H2D and final D2H copies). The benchmark uses sparse matrices with

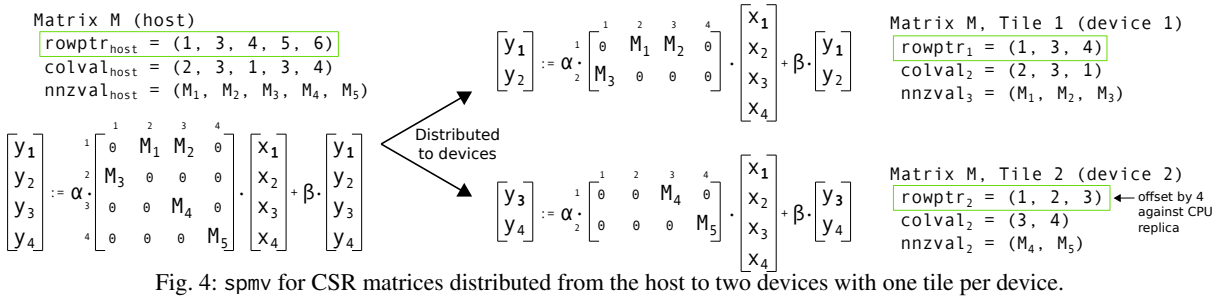

Fig. 4: spmv for CSR matrices distributed from the host to two devices with one tile per device.

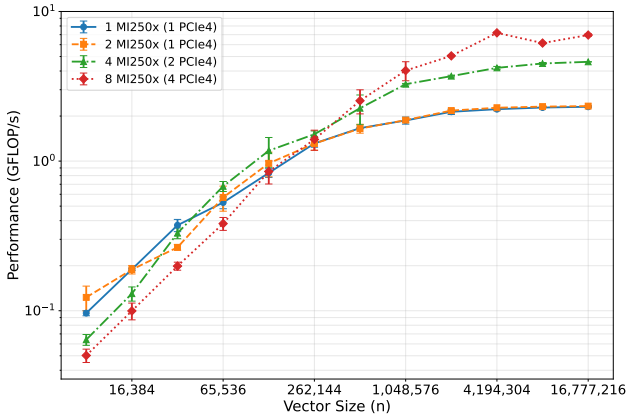

Fig. 5: Performance of an axpy on AMD-JLSE using 64-bits precision.

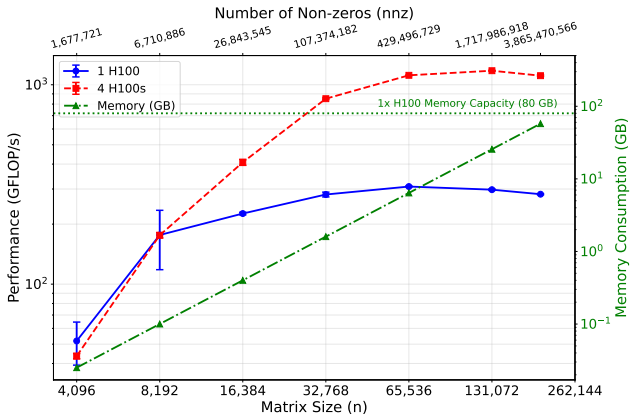

Fig. 6: Performance of an spmv on NV-JLSE using 64-bits precision.

non-zero elements generated randomly uniformly with a 10% density. Thus, by increasing n, we increase the amount of work. For $n \leq 2^{13}$ (about 800 nnz per line), we observe no performance improvement in increasing the number of GPUs. The reason is that spmv does not saturate streaming multiprocessors (SMs) units of a single GPU: there is no sufficient work for scaling. Pn the rightmost point (about 14,700 nnz per line), we saturate the memory capacity and SMs of a single GPU. In this case, we observe $3.5\times$ speedup by using all GPUs.

4.1.3 BLAS 3 - gemm. Figure 7 reports performance on a gemm scaling from 1 to 8 Xe Stacks on Aurora. We set $\alpha = 1$ and $\beta = 0$ to compute only $C := A.B$ with square matrices ($m = n = k$). Timings only include the computation (i.e., not the initial H2D and

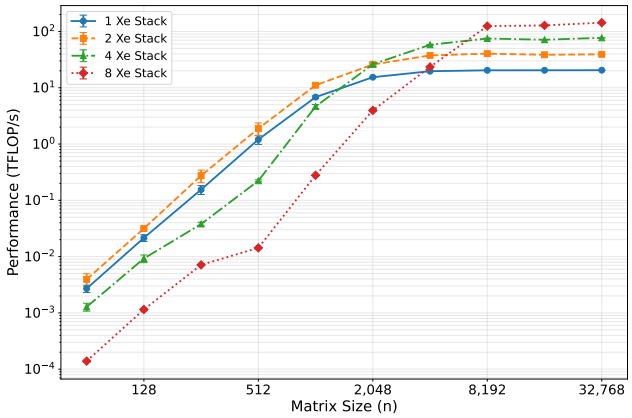

Fig. 7: Execution time of a gemm on Aurora using 32-bits precision.

final D2H copies). For small matrices, the lack of work does not allow to improve performance. However, for sufficiently large matrices ($n \geq 4,096$), we observe perfect scalability reaching theoretical bounds of about 20TFlop/s per Xe Stack [10].

4.1.4 Conclusion. This series of experiments shows that given sufficient work, XK.BLAS enables the automatic scaling of a CPU BLAS routine to multiple GPUs. They also illustrates its portability by running on all three major GPU vendors.

4.2 XK.BLAS as a Krylov.jl backend

Following our initial motivation, we evaluate XK.BLAS as a backend for Krylov.jl. We report results on the NV-JLSE system of the conjugate gradient solver. Other solvers exhibited similar performance trends. In the following, we first evaluate single-GPU performance and multi-GPU performance of Krylov.jl using XK.BLAS as a backend. We use two datasets:

—Three matrices (`nasarb`, `pwtk`, `Queen_4147`) from the Suite Sparse Matrix Collection (SSMC) [14]—for realistic test cases. These matrices are all symmetric positive definite, and cover different size ranges (see Table 4). The largest matrix (`Queen_4147`) still fits in the memory of a single H100 GPU.

—Three pseudo-random sparse matrix generators (M_1, M_2, M_3) representative of different scientific problems[3]. These matrices allow the exploration of large symmetric positive definite matrix sizes that would not fit on a single GPU. They also allow the exploration of different sparsity patterns—that we illustrate on visualizations in Appendix A.

[3]Claude Sonnet 4.5 [6] was used to create the matrix generators.

Table 2. : Pseudo-random matrices and performance obtained on NV-JLSE on Krylov.jl Conjugate Gradient implementation. t_{native} and $t_{xk.blas}$ are average times to execute a single iteration of the method, respectively, using the existing native cuBLAS backend of Krylov.jl, and our backend built upon XK.BLAS. Timings reported are the average execution time over iterations, and only include computation and D2D transfers: the initial H2D (to move matrices and vectors to GPUs) and the final D2H (to write-back the solution) are not included. We also report speedups against running XK.BLAS on a single GPU.

		Sizes			$\mathbf{t_{native}}$	$\mathbf{t_{xk.blas}}$			
Matrix	**Problem**	n	nnz	nnz/n	**1 GPU**	**G = 1 GPU**	**with $\mathcal{S}_3$**	**G = 2 GPUs**	**G = 4 GPUs**
M_1	Heat equation 7 point-stencil	287,496,000	2,009,858,400	7	$23.4ms$	$23.2ms$	no	$22.4ms$ (1.0×)	$21.3ms$ (1.1×)
							yes	$14.3ms$ (1.6×)	$8.4ms$ (2.8×)
M_2	Edge-element Maxwell Equation	22,473,360	468,728,904	21	$3.7ms$	$3.7ms$	no	$3.0ms$ (1.2×)	$2.8ms$ (1.3×)
							yes	$2.7ms$ (1.4×)	$2.3ms$ (1.6×)
M_3	Linear elasticity FEM of order 4	2,738,019	1,728,900,297	631	$7.4ms$	$7.4ms$	no	$5.6ms$ (1.3×)	$3.8ms$ (1.9×)
							yes	$5.5ms$ (1.3×)	$3.7ms$ (2.0×)

Table 3. : Compute and memory complexity of the Conjugate Gradient method with XK.BLAS using 1 tile per GPU. G is the number of GPUs, I is the number of iterations the solver performed, n is the number of rows and columns of the matrix, nnz is the total number of non-zero values in the matrix, and S is the size of type in byte (i.e., float is 4, double is 8, etc.).

Compute		Memory					
Amount (FLOP)	**# Launch**	**Amount (bytes)**			**# Launch**		
		H2D	D2H	P2P	H2D	D2H	P2P
$O((nnz+n)*I)$	$7*G*I$	$(nnz+n)*S$	$n*S$	$n*I*S$	2	1	$G*I$

Table 4. : SSMC matrices and performance obtained on NV-JLSE on Krylov.jl Conjugate Gradient implementation. t_{native} and $t_{xk.blas}$ are average times to execute a single iteration of the method, respectively, using the existing cuBLAS backend of Krylov.jl, and our backend built upon XK.BLAS using a single GPU ($G = 1$).

Name	**m=n**	**nnz**	**nnz/n**	$\mathbf{t_{native}}$	$\mathbf{t_{xk.blas}}$
nasasrb	54,870	2,677,324	49	$118 \pm 4\mu s$	$369 \pm 1\mu s$
pwtk	217,918	11,524,432	53	$162 \pm 2\mu s$	$415 \pm 3\mu s$
Queen_4147	4,147,110	316,548,962	76	$1.8 \pm 0ms$	$1.9 \pm 0.1ms$

4.2.1 Theoretical Complexities. Table 3 reports the computational and memory complexity of the conjugate gradient solver used throughout this section. For sufficiently many iterations ($I \gg 1$), the cost of the initial host-to-device (H2D) and final device-to-host (D2H) transfers is amortized, and the execution time is bounded either by the number of floating-point operations (FLOPs) or by peer-to-peer (P2P) memory transfers. The resulting arithmetic intensity is $O(\frac{nnz}{n})$, where nnz/n is the average number of non-zero elements per row. Additionally, the aggregated system peak performance (120 TFLOP64/s) [21] greatly exceeds its memory bandwidth (1.0 TB/s). Consequently, performance should scale with the number of GPUs (G) only if $nnz \gg n$; otherwise, the execution is limited by P2P memory copies.

4.2.2 Single-GPU Performance. We evaluate Krylov.jl performance on a single GPU using SSMC matrices. Table 4 compares the performance of XK.BLAS with the default cuBLAS native backend.
On the nasarb and pwtk matrices, we observe performance degradation using XK.BLAS against the original backend. We profiled execution with NVIDIA's nsys tool and XKRT built-in metrics (XKRT_STATS=1). Among the $118\mu s$ per iteration on the nasasrb matrix, only $33\mu s$ actually corresponds to the GPU computation that is partially overlapped with $97\mu s$ of device operations overheads (kernel launches, memory copy launches, and synchronizing). Similar overheads were observed on other matrices, as they do not depend on the input (i.e., the same operations are launched, but on different vectors/matrices). With XK.BLAS, we observed higher management overheads than native cuBLAS due to (1) its tasking abstractions that add constant overhead for each operation, and (2) its internal asynchronous progression engine that records a CUDA event per kernel launch (cuEventRecord)—while the "native" version synchronously waits for the kernel completion on the calling threads, reducing synchronization overheads.
For the Queen_4147 matrix, the workload is higher, and these overheads are hidden, leading to no performance degradation.

4.2.3 Multi-GPU Performance. We evaluate the scaling of the Krylov.jl solver on multiple GPUs using the XK.BLAS backend. Table 2 reports the pseudo-random sparse matrices we used to saturate the memory of a single H100 GPU. The table reports average times to execute a single iteration on a single GPU and multiple GPUs, using the original native cuBLAS backend and XK.BLAS. When using XK.BLAS on multiple GPUs, it reports times with and without the optimization step $\mathcal{S}_3$ introduced in section 3.3.
Without $\mathcal{S}_3$, speedups are low with 1.1×, 1.3× and 1.9× respectively for matrices M_1, M_2 and M_3 when scaling from 1 to 4 GPUs. With $\mathcal{S}_3$, however, we observe an improved performance scaling for all matrices. For instance on M_1, enabling $\mathcal{S}_3$ leads to 2.8× speedup against the initial 1.1×. By profiling with XKRT built-in metrics, we observe significantly fewer P2P transfers when $\mathcal{S}_3$ is enabled, with $20.0MB$ transferred on average per iteration against $6.4GB$ without $\mathcal{S}_3$. The reason for this P2P decrease stems from the matrix sparsity: M_1 is 7-diagonal. When distributing an spmv ($y := A.x$) to multiple devices, $\mathcal{S}_3$ reduces the amount of bytes copied to devices' local replicas of x, as explained in section 3.3.
These results validate theoretical complexities of Table 3. The greater nnz/n (i.e., the more non-zero elements there are per line), the better the performance scales with the number of GPUs. If nnz is too small, then there is not sufficient workload to amortize the extra P2P communications induced by multi-GPU parallelization. Enabling $\mathcal{S}_3$ can reduce P2P communication and improve scalability if the matrix sparsity pattern allows it.

5. Conclusion

This paper introduced XK.BLAS, the BLAS module of the package XK.jl. XK.BLAS provides a composable, multi-GPU BLAS API in Julia, portable across the three major GPU vendors (AMD, Intel, NVIDIA). It is built on top of the XKRT tasking runtime system and the XKBlas multi-GPU BLAS C++ library. Unlike existing linear algebra packages, XK.BLAS automates the parallelization of CPU-based linear algebra programs to the scale of a compute node equipped with multiple GPUs, while managing memory and task scheduling transparently.
The primary motivation for XK.BLAS was to provide a multi-GPU backend for Krylov.jl, a collection of (block) Krylov solvers. XK.BLAS not only enables multi-GPU execution of solvers in

Krylov.jl but also improves portability: it abstracts available devices by manipulating host-memory objects (`Vector`, `Matrix`, `SparseMatrixCSR`) and automatically moving memory to executing devices. To implement this backend, we extended XKBlas with support for selected BLAS 1 and 2 routines (`axpy`, `dot`, `scal`, `gemv`, `spmv`, etc.). Evaluations show 2.5× slowdown on the smallest evaluated matrix (`nasasrb`) due to overhead induced by memory coherence automation. Overheads are fully amortized on large matrices (e.g., `Queen_4147`)—and we show up to 2.8× speedup when automatically scaling from 1 to 4 H100 GPUs within the same node. XK.BLAS also demonstrates portability by executing on three distinct multi-GPU architectures (MI250X, PVC, H100).
Future work within the XKRT and XK.jl ecosystem includes:

- **Enhanced BLAS and LAPACK support:** Extending XKBlas to implement additional BLAS and LAPACK routines will automatically propagate improvements to XK.jl and its BLAS module. For example, `spmv` is not yet available on AMD / Intel GPUs, and several LAPACK routines are required to fully support Krylov.jl's block solvers.
- **Batching command submissions:** As shown in section *4.2.2*, the latency overheads of eagerly submitting GPU operations currently limit Krylov.jl performance for small matrices, due to frequent CPU–GPU synchronization. Thanks to its underlying macro-dataflow model (XKRT), XK.BLAS could support operation batching (i.e, submitting large CUDA/HIP Graphs, or Level Zero command lists) to reduce overheads. We are currently exploring this direction that would improve the performance of both Krylov.jl and MUMPS. However, we expect Krylov.jl source modifications, so that the entire solver is exposed as a task dependency graph. For instance, NVIDIA provides an example for Conjugate Gradient using CUDA graphs [4].
- **Embedding a kernel language:** XK.jl currently supports a prototype that falls back to Julia's embedded LLVM compiler for NVPTX. Integrating a kernel language within XK.jl, combined with its macro-dataflow model, could enable JIT kernel fusion and improve programmer productivity, similar in spirit to `pytorch.compile` [5].

Finally, from a Julia runtime perspective, we encountered challenges running Julia code via FFI on XKRT "foreign" threads, often leading to deadlocks or requiring workarounds [5]. We believe these issues stem from design choices in Julia's tasking runtime rather than fundamental language limitations, and plan to investigate more robust solutions in future work.

6. Acknowledgment

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

[4] https://github.com/NVIDIA/cuda-samples/blob/4f735616ba599fe93cc2c6c85dcb4369260f9643/Samples/4_CUDA_Libraries/conjugateGradientCudaGraphs/conjugateGradientCudaGraphs.cu#L304-L351

[5] https://github.com/JuliaLang/julia/issues/59640

7. References

[1] Ahmad Abdelfattah, Natalie Beams, Robert Carson, Pieter Ghysels, Tzanio Kolev, Thomas Stitt, Arturo Vargas, Stanimire Tomov, and Jack Dongarra. Magma: Enabling exascale performance with accelerated blas and lapack for diverse gpu architectures. *The International Journal of High Performance Computing Applications*, 38(5):468–490, 2024. doi:10.1177/10943420241261960. https://doi.org/10.1177/10943420241261960.

[2] Mayez Al-Mouhamed, Lutfi Firdaus, Ayaz H. Khan, and Nazeeruddin Mohammad. Spmv and bicg-stab sparse solver on multi-gpus for reservoir simulation. *Multimedia Tools and Applications*, 83(8):23563–23597, 2023. doi:10.1007/s11042-023-16185-0.

[3] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. MUMPS: A General Purpose Distributed Memory Sparse Solver. In Tor Sørevik, Fredrik Manne, Assefaw Hadish Gebremedhin, and Randi Moe, editors, *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, pages 121–130, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-70734-4_16.

[4] Petros Anastasiadis, Nikela Papadopoulou, Nectarios Koziris, and Georgios Goumas. Uncut-GEMMs: Communication-Aware Matrix Multiplication on Multi-GPU Nodes . In *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 143–154, Los Alamitos, CA, USA, September 2024. IEEE Computer Society. doi:10.1109/CLUSTER59578.2024.00020.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3620665.3640366.

[6] Anthropic. Claude sonnet 4.5, 2025. Large language model. Accessed January 30, 2026.

[7] Basic Linear Algebra Subprograms Technical (BLAST) Forum. Basic linear algebra subprograms (blas) technical forum standard. https://netlib.org/blas/blast-forum/blas-report.pdf, August 2001. Accessed: Nov. 04, 2024.

[8] Michael Bauer. Legion: programming distributed heterogeneous architectures with logical regions. 2014.

[9] Olivier Beaumont, Philippe Duchon, Lionel Eyraud-Dubois, Julien Langou, and Mathieu Vérité. Symmetric block-cyclic distribution: Fewer communications leads to faster dense cholesky factorization. In *SC22: International Conference for High Performance Computing,*

Networking, Storage and Analysis, pages 1–15, 2022. doi:10.1109/SC41404.2022.00034.

[10] Colleen Bertoni, Thomas Applencourt, Longfei Gao, and Ti Leggett. Millions of matrix-multiplications: Gemm variations on aurora. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 850–856, 2025. doi:10.1109/IPDPSW66978.2025.00135.

[11] Aurelien Bouteiller, Thomas Herault, Qinglei Cao, Joseph Schuchart, and George Bosilca. PaRSEC: scalability, flexibility, and hybrid architecture support for task-based applications in ECP. *The International Journal of High Performance Computing Applications*, 39(1):10943420241290520, 2024. doi:10.1177/10943420241290520. https://doi.org/10.1177/10943420241290520.

[12] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009. doi:10.1016/j.parco.2008.10.002.

[13] Intel Corporation. Intel® oneapi math kernel library (onemkl), 2025. Accessed: 2025-09-29.

[14] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. doi:10.1145/2049662.2049663.

[15] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems.

[16] Mark Gates, Ahmad Abdelfattah, Kadir Akbudak, Mohammed Al Farhan, Rabab Alomairy, Daniel Bielich, Treece Burgess, Sébastien Cayrols, Neil Lindquist, Dalal Sukkari, and Asim YarKhan. Evolution of the SLATE linear algebra library. *The International Journal of High Performance Computing Applications*, 39(1):3–17, 2025. doi:10.1177/10943420241286531. https://doi.org/10.1177/10943420241286531.

[17] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. pages 15–23, 07 2007. doi:10.1145/1278177.1278182.

[18] Thierry Gautier and João V. F. Lima. Xkblas: a high performance implementation of blas-3 kernels on multi-gpu server. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 1–8, 2020. doi:10.1109/PDP50117.2020.00008.

[19] David Krasowska. Reducing complexity in scalable numerical computing with cunumeric.jl. Poster presented at the 2025 Department of Energy Computational Science Graduate Fellowship (CSGF) Program Review, Apr 2025. Northwestern University, Evanston, IL.

[20] Alexis Montoison and Dominique Orban. Krylov.jl: A Julia basket of hand-picked Krylov methods. *Journal of Open Source Software*, 8(89):5187, 2023. doi:10.21105/joss.05187.

[21] NVIDIA Corporation. NVIDIA H100 GPU Datasheet. Online, 2025. Accessed: 2026-01-12.

[22] NVIDIA® CUDA™. Cuda cublas library - version 1.0. https://developer.download.nvidia.com/compute/cuda/1.0/CUBLAS_Library_1.0.pdf, June 2007. Accessed: Nov. 05, 2024.

[23] NVIDIA® CUDA™. cuBLAS – Using the cuBLASXt API. https://docs.nvidia.com/cuda/cublas/, 2024. Accessed: Oct. 21, 2024, Updated: Sep 30, 2024.

[24] AMD ROCm Team. rocblas: Amd's blas library for the rocm platform, 2025. Accessed: 2025-09-29.

[25] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2925426.2926256.

APPENDIX

A. Matrices

The following figures are visualization of non-zero elements from matrices similar to the one used in Table 2.
Remark: Sizes differ from our evaluations (Section 4) but the sparsity patterns are identical. We reduced the sizes to avoid rendering artifacts.

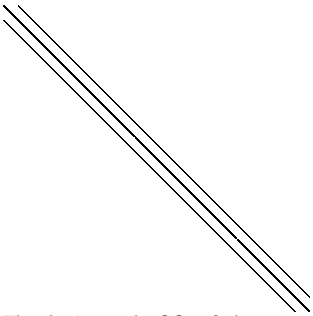

Fig. 8: A matrix M_1 of size $n = 9,261$ and $62,181$ non-zeroes

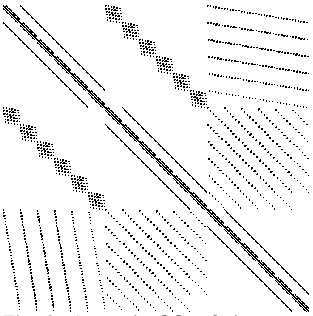

Fig. 9: A matrix M_2 of size $n = 540$ and $8,784$ non-zeroes

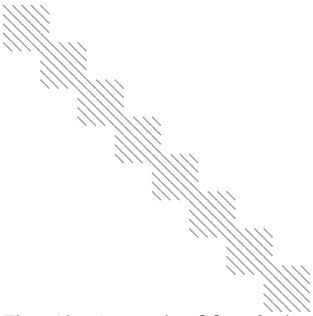

Fig. 10: A matrix M_3 of size $n = 107,811$ and $64,701,513$ non-zeroes