Composyx

Emmanuel AGULLO, Luc GIRAUD,
Arthur GOUINGUENET, Gilles MARAIT

Exa-MA Annual Meeting - January 20th 2026

Contents

Composyx in a nutshell

Composyx in a nutshell

Composyx is a C++ header-only linear algebra package.

- Scope: **sparse**, **dense**, **randomized** linear algebra
- Implemented in **C++20** with concepts (soon C++23).
- Provides a **high-level interface** for prototyping, close to Matlab or python.
- Possibility to dig deeper to low-level interfaces for performance.
- Literate programming in org-mode: check out our beautiful documentation.
- Supercomputers
  - Supports distributed memory parallelism
  - Task-based support
  - GPU-only implementation, currently only **CUDA**, compatible with distributed parallelism
- Experimental features for **variable accuracy** and **mixed-precision**.
- Many tutorials and examples to use **Composyx** for linear algebra or from an numerical application.

Installing Composyx

Header-only version

To test **Composyx** with minimal dependencies, a header-only implementation is sufficient.

```
git clone --recursive https://gitlab.inria.fr/composyx/composyx.git
COMPOSYX_INCLUDES=`realpath composyx/include`
TLAPACK_INCLUDES=`realpath composyx/tlapack/include`

g++ -std=c++20 -I${COMPOSYX_INCLUDES} -I${TLAPACK_INCLUDES} my_composyx_code.cpp
```

Header-only installation

- Relies on `<T>LAPACK` header-only BLAS/LAPACK implementation
- Perfect to test **Composyx** without any installation step
- Only a subset of **Composyx**
- Only for prototyping, no expectation for performance

Installation with guix

GNU guix is a cross-platform package manager with a strong focus on **software reproducibility**.

Composyx is available as a package in guix-science.

```
guix shell composyx -- composyx_driver_cg
```

guix installation

- When guix is available, deployment of an environment with **Composyx** and most of its important dependencies is **easy**, **fast** and **reliable**
- Environment is well controlled and reproducible for scientific experiments
- Possibility to use non-free HPC software (mkl, cuda, …)

Installation with composyx-superbuild

A **CMake** based automatic installation is available through composyx-superbuild. This is the preferred solution when guix is not available.

```
git clone https://gitlab.inria.fr/composyx/composyx-superbuild.git
cd composyx-superbuild
cmake -B build
cmake --build build/
```

Installation with composyx-superbuild

- Download and install with CMake **Composyx** and its dependencies automatically
- Basic dependencies must be installed already: CMake, MPI, blas, git, python…
- Possibility to download dependencies in a tarball and then deploy them without internet access on a supercomputer
- Possibility to choose which dependencies to use with **Composyx**

Other installation methods

Other methods are available to install **Composyx**.

Spack

A spack package of **Composyx** is available on the official repository.

Brew

For MacOS users, a brew package is available.

Manual installation

Composyx uses CMake and dependencies to be used are configurable. Manual installation is still a good solution for advanced users or when few dependencies are necessary.

Using Composyx

High level interface

Special operators in composyx

- `operator*` apply an operator to a vector: `y = A * x`
- `operator~` compute an inverse (or genealized inverse) of an operator `A_inv = ~A`
- `operator%` apply a genealized inverse operator to a vector (as the operator `\` in *Matlab*) `x = A % b` ⇔ `x = ~A * b`

BLAS/LAPACK-like functions

```
blas_kernels::axpy(u, v, Scalar{2}); // v <- 2 * u + v
```

High level functions returning tuples

```
auto [Q, R] = blas_kernels::qr(A);
auto [U, S, V] = blas_kernels::svd(A);
```

High level interface example

Call a BLAS operation

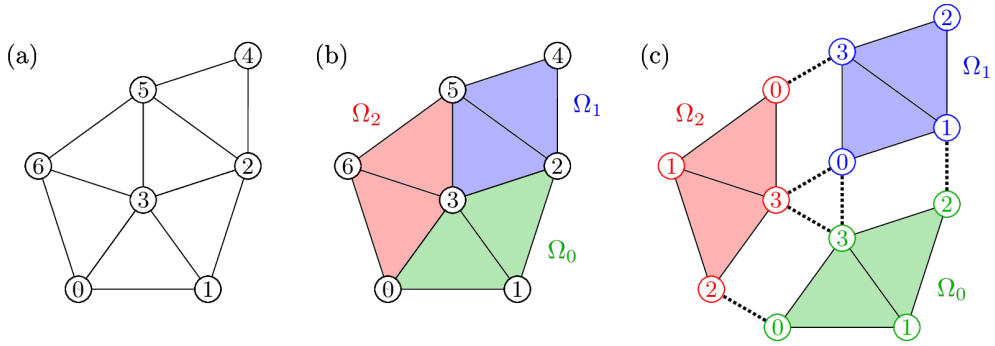
```
using namespace composyx;
using Vect = Vector<Scalar>;
using Mat = DenseMatrix<Scalar>;

const Vect u{1, 2, 3, 4};
Vect v{10, 20, 30, 40};
// v <- v + 2 * u

// ---
// Write directly
v += 2 * u;

// --- Or ---
// Use blas interface
blas_kernels::axpy(u, v, Scalar{2});
```

Solve a system

```
const Mat A({1, 2, 3,
             4, 5, 6,
             7, 8, 9}, 3, 3);
const Vect b{30, 20, 10};

// ---
// Instanciate solver manually
BlasSolver<Mat, Vect> solver(A);
Vect x = solver * b;

// --- Or ---
// Use the operator ~
auto solver = ~A;
Vect x = solver * b;

// --- Or ---
// Use operator %
Vect x = A % b;
```

Domain decomposition-like interface

K is the global sparse matrix representing the system to solve.



K global adjacency graph

	0	1	2	3	4	5	6
0	x	x		x			x
1	x	x	x	x			
2		x	x	x	x	x	
3	x	x	x	x		x	x
4			x		x	x	
5			x	x	x	x	x
6	x			x		x	x

K partitioned, **interior edges** are colored, *interface edges* are black

	0	1	2	3	4	5	6
0	x	x		x			x
1	x	x	x	x			
2		x	x	x	x	x	
3	x	x	x	x		x	x
4			x		x	x	
5			x	x	x	x	x
6	x			x		x	x

$K0$

	0	1	2	3
0	x	x		x
1	x	x	x	x
2		x	x	x
3	x	x	x	x

$K1$

	0	1	2	3
0	x	x		x
1	x	x	x	x
2		x	x	x
3	x	x	x	x

$K2$

	0	1	2	3
0	x	x		x
1	x	x	x	x
2		x	x	x
3	x	x	x	x

Domain decomposition-like interface

Specific datatypes are used for this purpose:

- `PartMatrix` to describe the matrices
- `PartVector` to describe the vectors
- `PartOperator` to use other operators (typically solvers)
- Dedicated **preconditioners** can be used on iterative solvers

Then **Composyx** solvers can be used transparently on those datatypes.

Solve a system

```
const PartMatrix<SparseMatrixCSR<Scalar>> A(/* ... */);
const PartVector<Vector<Scalar>> b{/* ... */};
using SparseDirectSolver = Mumps<SparseMatrixCSR<Scalar>, Vector<Scalar>>;
using Pcd = AdditiveSchwarz<decltype(A), decltype(B), SparseDirectSolver>;
ConjugateGradient<PartMatrix<SparseMatrixCSR<Scalar>>,
  PartVector<Vector<Scalar>>, Pcd> cg(A);
auto x = cg * b;
```

Special types in **Composyx** to handle vectors and matrices on a CUDA device:

On CPU	On GPU
`Vector<Scalar>`	`CudaVec<Vector<Scalar>>`
`DenseMatrix<Scalar>`	`CudaDense<DenseMatrix<Scalar>>`
`SparseMatrixCSR<Scalar>`	`CudaSparse<SparseMatrixCSR<Scalar>>`

Then the code is almost the same as for CPU.

```
using namespace composyx;
using Vect = CudaVec<Vector<Scalar>>;
using Mat = CudaDense<DenseMatrix<Scalar>>;
const Vect u{1, 2, 3, 4};
Vect v{10, 20, 30, 40};
// v <- v + 2 * u
v += 2 * u;
// --- Or ---
cuda_blas_kernels::axpy(u, v, Scalar{2});
```

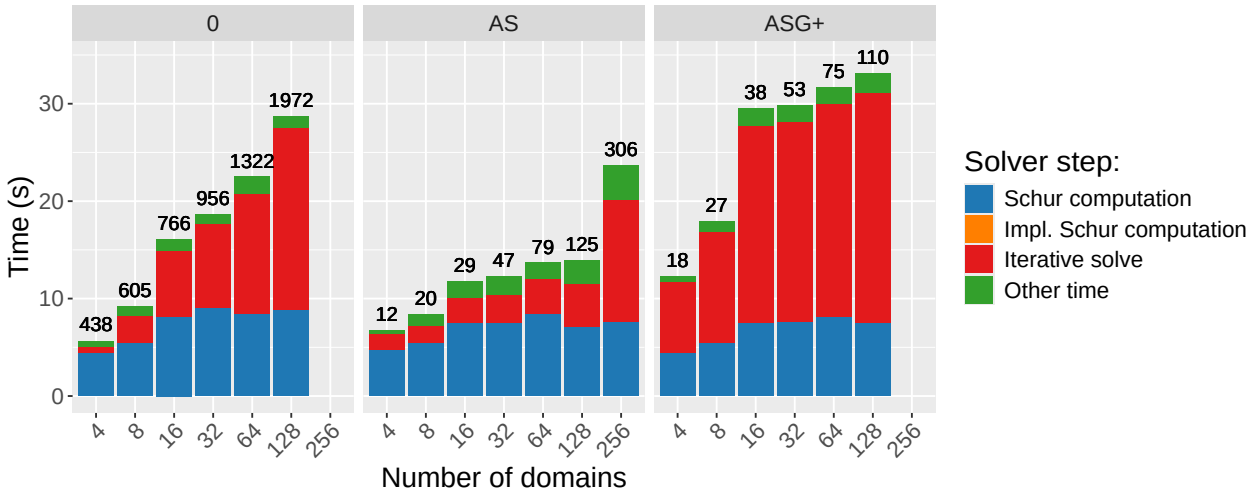
```
const PartMatrix<CudaSparse<SparseMatrixCSR<Scalar>>>
A(/* ... */);
const PartVector<CudaVec<Vector<Scalar>>> b{/* */};
using SparseDirectSolver = CudaSparseSolver<Scalar>>;
using Pcd = AdditiveSchwarz<decltype(A), decltype(B),
SparseDirectSolver>;

ConjugateGradient<decltype(A),decltype(B),Pcd> cg(A);
auto x = cg * b;
```

Results on GPU

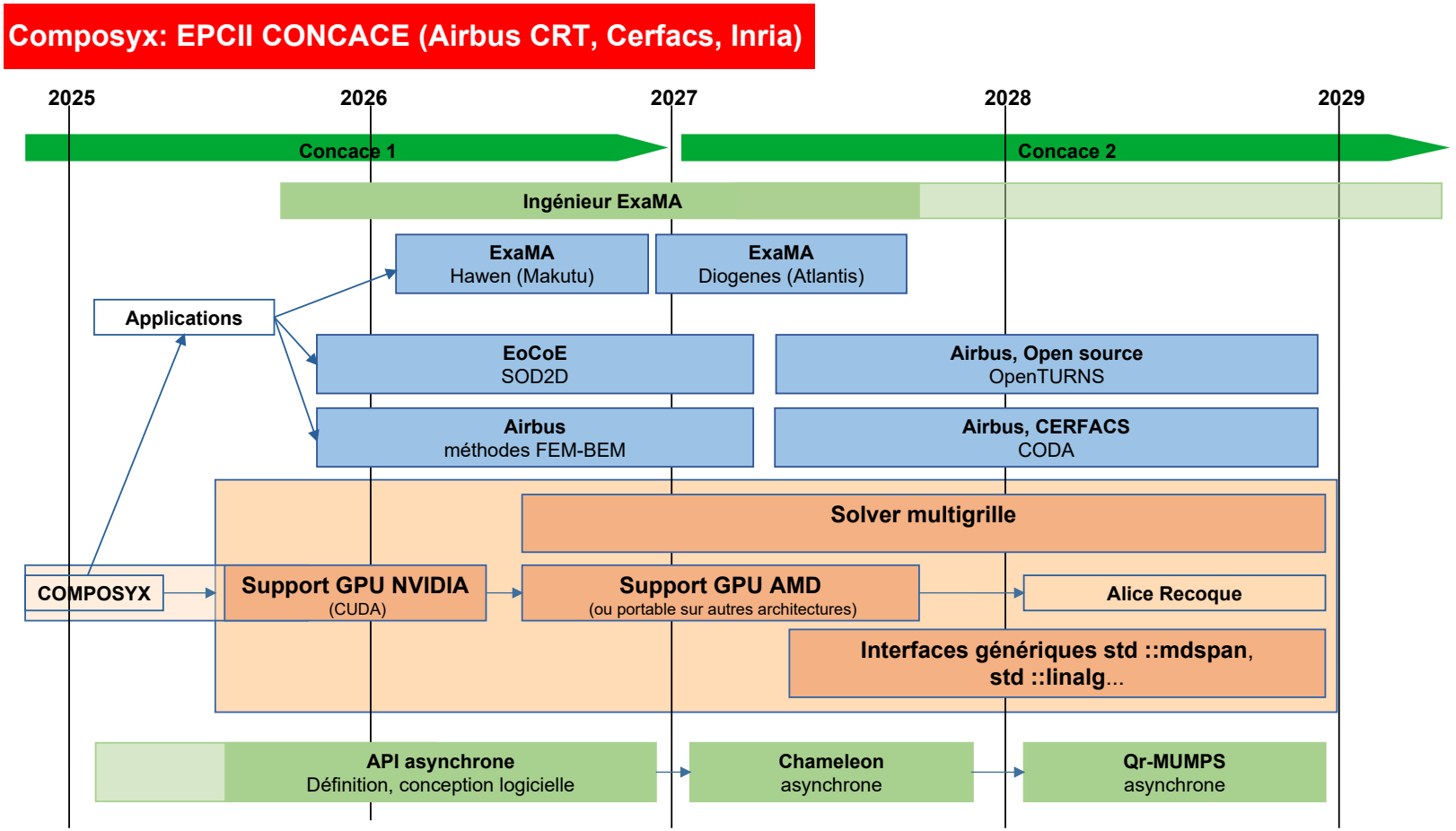
Results on Jean-Zay v100 partition, **weak scaling**, with 1 GPU per subdomain and 4 GPU per node.

Preconditioners:

- 0: no preconditioner
- AS: Additive Schwarz
- ASG+: Additive Schwarz + geneo additive coarse correction



Composyx roadmap 2025 - 2029



Thank you for your attention

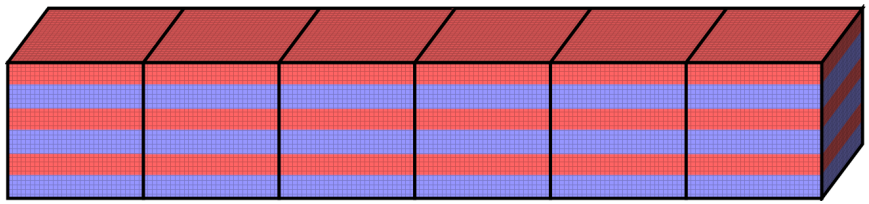
Useful links

- Repository: https://gitlab.inria.fr/composyx/composyx
- Documentation: https://composyx.gitlabpages.inria.fr/composyx/
- Tutorial: https://composyx.gitlabpages.inria.fr/composyx/main/tutorial/tuto.html
- Examples: https://gitlab.inria.fr/composyx/Examples

Test case

Stationary heterogeneous diffusion equation (or Darcy equation) in a 3D stratified medium:

$$\nabla \cdot (k \nabla u) = 1$$



- each subdomain consists in $60 \times 60 \times 60 =$ **216 000** unknowns;
- interfaces between two adjacent subdomains have $60 \times 60 =$ **3 600** unknowns;
- benchmark in **weak scaling** (subdomains are stacked on along an axis)
- we use **one subdomain per GPU**, $n_{\text{sd}} = n_{\text{GPU}}$

n_{sd}	4	8	16	32	64	128	256
Size K	853 200	1 702 800	3 402 000	6 800 400	13 597 200	27 190 800	54 378 000
Size S	10 800	25 200	54 000	111 600	226 800	457 200	918 000

Test case

Environment

- gcc-14.2.0
- intel-oneapi-mkl 2023.1 (unused?)
- openmpi 4.1.8
- cuda toolkit (12.8 to check)
- cudss-0.7.1.4

Test case

- Preconditioners:
  - 0: no preconditioner
  - AS: Additive Schwarz
  - ASG+: Additive Schwarz + geneo additive coarse correction
- Number of eigenvectors for ASG+ preconditioner $n_{\text{ev}} = 3$
- Stopping criterion: $\frac{\|b-Ax\|}{\|b\|} < 10 - 8$

Hardware

Jean Zay supercomputer, copyright Photothèque CNRS/Cyril Frésillon

We run on the *Jean Zay* supercomputer, **v100** partition:

- 396 nodes with **4 GPUs per node**:
  - 2 CPUs Intel Xeon Gold 6248 (20 cores at 2.5 GHz), hence 40 cores per node
  - 192 GB of memory per node
  - 270 nodes with 4 GPUs Nvidia Tesla V100 SXM2 32 GB
  - Max reservation is 256 GPU (64 nodes)
  - Interconnect: Omni-path 100 series, 25 GB/s