

XianzhiKrylov 26

用户手册

C API 使用与参考

版本：26.48

适用平台：NVIDIA CUDA 12.4 / 12.9 / 13.1

修订日期：2026 年 11 月 22 日

目录

1 简介 **1**
1.1 求解方法与数据 . . . 1
1.2 如何阅读本手册 . . . 1
1.3 软件包中的文件 . . . 2

2 安装与首次运行 **3**
2.1 平台、软件版本与架构支持 . . . 3
2.2 准备环境 . . . 3
2.3 解压软件包 . . . 4
2.4 放置授权文件 . . . 4
2.5 编译与运行 . . . 4
2.6 运行库路径 . . . 4
2.7 使用 GitHub 样例 . . . 5
2.8 编译自己的 C 程序 . . . 5

3 调用流程 **6**
3.1 计算对象与数据对象 . . . 6
3.2 一次完成或分阶段执行 . . . 6
3.3 数组的保存与释放 . . . 6
3.4 重复使用准备结果 . . . 7
3.5 查看运行时间 . . . 7

4 数据类型与存储格式 **8**
4.1 精度与 C 数据类型 . . . 8
4.2 填写矩阵描述 . . . 8
4.3 CSR：按行存储非零元素 . . . 8
4.4 其他稀疏格式 . . . 9
4.5 一个向量与多个向量 . . . 10
4.6 DD 输入 . . . 10

5 函数参考 **11**
5.1 `xianzhikrylov_create()` . . . 11
5.2 `xianzhikrylov_config_set()` . . . 11
5.3 `xianzhikrylov_matrix_create()` . . . 12
5.4 `xianzhikrylov_vector_create()` . . . 12
5.5 `xianzhikrylov_multiple_vector_create()` . . . 13
5.6 `xianzhikrylov_execute()` . . . 14
5.7 `xianzhikrylov_print_info()` . . . 14
5.8 `xianzhikrylov_matrix_destroy()` . . . 14
5.9 `xianzhikrylov_vector_destroy()` . . . 15
5.10 `xianzhikrylov_destroy()` . . . 15
5.11 分阶段执行与连续求解 . . . 15

6 样例与求解配置 **16**
6.1 从随包样例开始 . . . 16
6.2 本样例的配置文件 . . . 17

6.3 选择方法和预条件器 . . . 18
6.4 方法、预条件器与 Level . . . 19
6.5 详细预条件器与参数 . . . 19
6.6 分阶段调用与数据更新 . . . 20
6.7 初值 . . . 21
6.8 分别学习各精度 . . . 21
6.9 多右端项：double 样例 . . . 21
6.10 与 LinglongAMG 配合 . . . 21

7 输出、验证与常见问题 23
7.1 如何判断求解成功 . . . 23
7.2 残差与容差 . . . 23
7.3 可选真实残差验收 . . . 23
7.4 独立验证 . . . 24
7.5 返回状态 . . . 24
7.6 常见问题 . . . 24

附录 A 样例清单与矩阵文件 25
A.1 按求解方法选择样例 . . . 25
A.2 精度样例：每个只有一个右端项 . . . 25
A.3 多右端项：单独的 double 样例 . . . 25
A.4 如何编译 . . . 26
A.5 矩阵文件与数值生成 . . . 26

附录 B 文档与技术支持 27
B.1 在哪里查找说明 . . . 27
B.2 软件包与授权 . . . 27
B.3 联系技术支持 . . . 27

附录 C 版本日志 28
XianzhiKrylov 26.48 . . . 28
XianzhiKrylov 26.46 . . . 28
XianzhiKrylov 26.45 . . . 29
XianzhiKrylov 26.44 . . . 29
XianzhiKrylov 26.43 . . . 29
XianzhiKrylov 26.42 . . . 29

1 简介

XianzhiKrylov 是一个使用 GPU 求解稀疏线性方程组的软件库。给定矩阵 A 和右端项 B，它计算满足下式的解 X：

$$AX = B.$$

这里 A 是 $n \times n$ 方阵。“稀疏”表示矩阵中的大部分元素为零，程序只需存储有用的元素。B 为已知右端项，X 为待求的解。B 和 X 可以包含一列或多列；多列右端项对应同一个系数矩阵 A。

应用通过 C 函数传入数据，用 JSON 配置文件选择求解方法、预条件器和精度要求。预条件器是一种辅助计算，用来改善迭代方法的收敛速度。各 GPU 平台的软件包使用统一的 C 接口；安装目标 GPU 对应的软件包即可。

1.1 求解方法与数据

类别	内容
迭代方法	CG、PCG、FCG、BiCG、BiCGStab、GMRES、FGMRES、GCR。
预条件器	Jacobi、ILU(0)、IC(0)、块 Jacobi、DILU、DIC、多色块 DILU。
矩阵格式	CSR、BSR、MSR、LDU。
数值类型	FP64（`double`）、FP32（`float`）、复数 FP64、复数 FP32、Double-Double（DD）。
外部 AMG	可调用 LinglongAMG 作为代数多重网格预条件器。
输入与结果	应用提供 A、B 和存放 X 的数组；求解完成后直接从 X 数组读取结果。

1.2 如何阅读本手册

本手册适用于 XianzhiKrylov 26.48 的 CUDA 软件包。第一次使用可先按第 2 章运行独立 Samples 样例，再阅读第 6 章的代码和配置。将库接入自己的程序时，重点查看第 4 章的数据格式和第 5 章的函数参数。

等宽字体表示命令、路径、函数和配置字段。解释性正文中的 `HOST`、`DEVICE`、`SUCCESS` 等简称省略了 `XIANZHIKRYLOV_` 前缀；C 代码使用完整名称。“主机内存”指 CPU 内存，“设备内存”指 GPU 显存。

1.3 软件包中的文件

解压后，主要文件位于 XianzhiKrylov/ 目录中：

```
XianzhiKrylov/
  README.md                  # 安装与运行说明
  THIRD_PARTY_NOTICES.txt     # 第三方组件的许可说明
  include/                   # C 程序使用的头文件
    XianzhiKrylovAPI.h        # 函数与返回状态
    XianzhiKrylovData.h       # 数据类型与矩阵描述
  lib/                       # 共享库及其运行文件
    libXianzhiKrylov.so
    ...
  xianzhikrylov_samples/     # 商城下载时合入的独立C样例
    README.md                # 样例总览与运行步骤
    Makefile                 # 编译全部样例
    common.mk                # 公共编译规则
    common/                  # 读取数据与验证结果的辅助代码
    matrices/                # 多个样例共用的矩阵文件
    docs/                    # 产品手册与常见问题
    17_fgmres_ilu0/           # 一个独立样例
      sample.c               # 程序
      config.json            # 迭代法和预条件器配置
      Makefile               # 编译、运行、清理命令
      README.md              # 本样例的说明
    ...
```

软件包与授权文件可从 https://store.solved-tech.com/ 下载。授权文件的放置方法见第2.4节。

2 安装与首次运行

2.1 平台、软件版本与架构支持

本手册的 CUDA 发行包面向 NVIDIA 平台、Linux x86_64，要求 glibc 不低于 2.35。以下为各包实际编译目标；原生 SM 与 PTX 目标分别列出。

CUDA	SM	PTX
12.4	sm_50, sm_52, sm_53, sm_60, sm_61, sm_62, sm_70, sm_72, sm_75, sm_80, sm_86, sm_87, sm_89, sm_90, sm_90a	compute_90
12.9	sm_50, sm_52, sm_53, sm_60, sm_61, sm_62, sm_70, sm_72, sm_75, sm_80, sm_86, sm_87, sm_89, sm_90, sm_90a, sm_100, sm_100a, sm_101, sm_101a, sm_103, sm_103a, sm_120, sm_120a, sm_121, sm_121a	compute_100f, compute_101f, compute_103f, compute_120f, compute_121f
13.1	sm_75, sm_80, sm_86, sm_87, sm_88, sm_89, sm_90, sm_90a, sm_100, sm_100a, sm_103, sm_103a, sm_110, sm_110a, sm_120, sm_120a, sm_121, sm_121a	compute_100f, compute_103f, compute_110f, compute_120f, compute_121f

带 a 或 f 后缀的目标具有相应架构或架构族限制，不能视为对所有更高 SM 的通用支持。PTX 执行需要驱动支持对应 PTX 版本。请使用兼容所选 CUDA 运行时的 NVIDIA 驱动。C++ 运行库需提供：CUDA 12.4 的 GLIBCXX_3.4.29，CUDA 12.9/13.1 的 GLIBCXX_3.4.32；三包均需 CXXABI_1.3.9 及 GOMP_4.0。

本章使用独立 Samples 中的`17_fgmres_ilu0`样例。商城下载时可附带`xianzhikrylov_samples/`；也可从第2章末尾列出的 GitHub 仓库取得。将样例仓库的完整路径保存为`SAMPLES_ROOT`：

```
export SAMPLES_ROOT=/absolute/path/to/xianzhikrylov-samples
```

2.2 准备环境

需要 Linux 计算机、GPU 对应的工具包、XianzhiKrylov 软件包及其授权文件，以及 C 编译器和 GNU Make。Make 用来按 `Makefile` 中的规则编译和运行程序。

在终端输入以下命令，查看系统和编译工具：

```
uname -m
getconf GNU_LIBC_VERSION
cc --version
make --version
```

`x86_64` 或 `aarch64` 表示 CPU 架构；`glibc` 是 Linux 系统使用的基础运行库。选择包名中 CPU 架构、工具包版本与当前环境相符，且满足其 glibc 要求的软件包。CUDA、DTK 等工具包的版本要求见所下载软件包的 `README.md`。

若提示 `cc` 或 `make` 找不到，Ubuntu/Debian 用户可运行：

```
sudo apt update
```

```
sudo apt install build-essential
```

其他 Linux 系统可通过其软件管理工具安装；在集群上可请管理员提供编译环境。集群上的 GPU 程序应在分配了 GPU 的计算节点运行。

2.3 解压软件包

先进入存放下载文件的目录。下面的 PACKAGE.tar.gz 代表实际下载的压缩包名，请替换后执行：

```
tar -xzf PACKAGE.tar.gz
cd XianzhiKrylov
export XK_ROOT="$(pwd)"
```

XK_ROOT 保存当前软件包的完整路径。后续命令请在同一个终端中执行。检查解压结果：

```
ls "$XK_ROOT/include/XianzhiKrylovAPI.h"
ls "$XK_ROOT/lib/libXianzhiKrylov.so"
```

2.4 放置授权文件

CUDA 26.48 支持以下授权文件位置：

- 当前用户目录~/.solved-tech/xianzhikrylov/，可保留下载文件的.key 文件名。
- 系统目录/etc/solved-tech/xianzhikrylov/，由管理员放置.key 文件。
- 兼容旧版的 SDK 包根目录license.key 或licenses/*.key。

如需为当前用户安装，将YOUR_LICENSE.key 替换为实际文件名：

```
install -d -m 700 "$HOME/.solved-tech/xianzhikrylov"
install -m 600 YOUR_LICENSE.key "$HOME/.solved-tech/xianzhikrylov/YOUR_LICENSE.key"
```

2.5 编译与运行

```
cd "$SAMPLES_ROOT/17_fgmres_ilu0"
make LIBRARY_ROOT="$XK_ROOT"
make run LIBRARY_ROOT="$XK_ROOT"
```

make 生成名为 sample 的程序；makerun 使用本目录的 config.json 和共用矩阵文件运行它。程序在 CPU 内存中生成 A、b 和初值 x，库负责 GPU 计算所需的数据准备。

正常结束时，可看到求解统计和独立残差检查，例如：

```
XianzhiKrylov status=1 converged=1 iterations=...
Independent residual RHS 1: relative_l2=... absmax=... PASS
```

这里的省略号代表本次实际数值。PASS 表示样例按原始 A、b 对结果进行了独立检查。在运行命令之后立刻输入 echo$?，输出 0 表示程序成功结束。

2.6 运行库路径

样例的 Makefile 会找到 XianzhiKrylov 库。如果运行时提示某个工具包共享库找不到，可在样例目录执行：

```
ldd ./sample
```

输出中的 notfound 表示该文件未找到。将包含该文件的工具包目录传给 GPU_RUNTIME_LIB_DIR。例如，该文件确实安装在 /usr/local/cuda-12.4/lib64 时：

```
make run GPU_RUNTIME_LIB_DIR=/usr/local/cuda-12.4/lib64
```

其他安装位置请使用实际目录。该设置帮助程序在运行时找到依赖库。

确认运行成功且不再需要已编译的程序时，可清理它：

```
make clean
```

2.7 使用 GitHub 样例

GitHub 仓库为 solved-tech-dev/xianzhikrylov-samples。在网页中选择 Code、DownloadZIP，将下载的 ZIP 解压。打开解压得到的 xianzhikrylov-samples-main/ 目录。将软件包解压到该目录的 libraries/ 下，使其头文件位于 libraries/XianzhiKrylov/include/，并按第2.4节安装授权文件。随后可在仓库的样例目录执行同样的 make、makerun 命令。例如，在解压后的仓库目录打开终端，执行：

```
cd 17_fgmres_ilu0
make
make run
```

已有软件包位于其他目录时，可沿用前面设置的路径：

```
make LIBRARY_ROOT="$XK_ROOT"
make run LIBRARY_ROOT="$XK_ROOT"
```

2.8 编译自己的 C 程序

已有程序 my_app.c 可包含 XianzhiKrylovAPI.h，并链接 libXianzhiKrylov.so。在该源文件所在目录中执行：

```
cc -std=c11 my_app.c -o my_app \
  -I"$XK_ROOT/include" -L"$XK_ROOT/lib" \
  -Wl,-rpath,"$XK_ROOT/lib" -Wl,--allow-shlib-undefined \
  -lXianzhiKrylov -lm
./my_app
```

输出程序名为 my_app。-I 指定头文件目录，-L 和 -lXianzhiKrylov 指定链接的库，-rpath 指定运行时查找本库的目录。授权文件按第2.4节放置。config_set() 中使用相对配置路径时，该路径相对于运行程序时所在的目录。若工具包运行库需要额外路径，将实际目录加入 LD_LIBRARY_PATH 后再运行；上节的 GPU_RUNTIME_LIB_DIR 是样例 Makefile 使用的变量。

3 调用流程

3.1 计算对象与数据对象

应用会创建三个种类的对象：

- 计算对象 handle：保存设备编号、求解配置、准备结果和统计信息，通常称为句柄。
- 矩阵对象 matrix：记录 A 的尺寸、格式和数组位置。
- 向量对象：分别记录 b 和 x 的数组位置。

这些对象由创建函数返回。应用保存它们，并将它们传给后续函数。

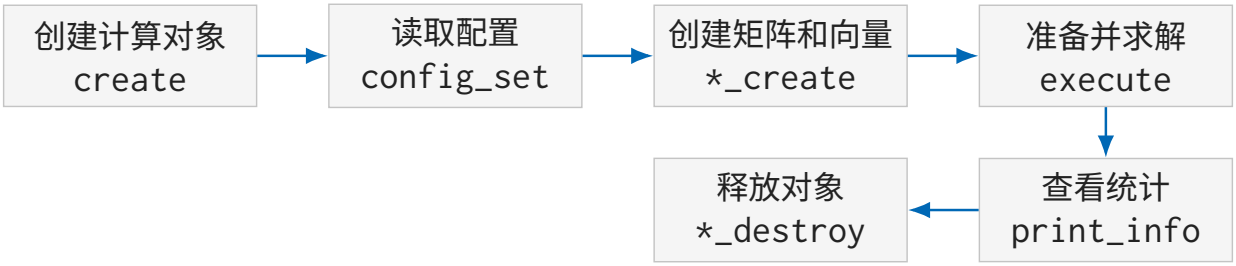

图 1: 一次求解的调用顺序。

3.2 一次完成或分阶段执行

xianzhikrylov_execute() 的参数依次为计算对象、阶段、A、x、b。第一次使用时，选择 XIANZHIKRYLOV_PHASE_ALL 即可一次完成准备和求解。

阶段	工作内容
XIANZHIKRYLOV_PHASE_ANALYSIS	根据矩阵中元素的位置进行结构分析；x、b 填 NULL。
XIANZHIKRYLOV_PHASE_NUMERIC_PREPARE	根据矩阵数值准备预条件器；x、b 填 NULL。
XIANZHIKRYLOV_PHASE_SOLVE	使用已准备的数据求解；传入 A、x、b。
XIANZHIKRYLOV_PHASE_ALL	依次完成上述三个阶段，并把解写入 x。

分析、数值准备成功返回 SUCCESS；求解满足停止条件返回 CONVERGED。多右端项求解需每个结果均满足停止条件。

3.3 数组的保存与释放

最直接的用法是将 A、b、x 放在 CPU 内存中，填写 XIANZHIKRYLOV_HOST。库会准备 GPU 计算所需的数据，并在求解返回时将最终解写入应用提供的 x 数组。

NVIDIA 和海光软件包还接受 GPU 显存中的数组，填写 XIANZHIKRYLOV_DEVICE。数组所在 GPU 应与计算对象选择的设备一致；库按所选方法完成必要的数据整理。

矩阵和向量对象保存数组位置，数组仍由应用管理。使用这些对象期间，应保持数组有效。先结束求解、销毁相关对象，再释放数组。销毁计算对象会释放库分配的计算工作区。

3.4 重复使用准备结果

当应用连续求解相近的系统时，可以按以下规则复用：

下一次求解有什么变化	需要执行的阶段
只更新 b 或初值 x	SOLVE。
A 中元素的位置不变，只更新数值	NUMERIC_PREPARE，再 SOLVE。
A 的结构改变，或换了矩阵对象	ANALYSIS、NUMERIC_PREPARE、SOLVE。
希望一次完成所有工作	ALL。

只更新数值时，保留矩阵对象及其数组地址，在原数组中写入新数值。需要更换配置时，创建新的计算对象并重新设置配置。同一计算对象的调用按顺序执行；并行求解独立系统时，分别创建计算对象和数据对象。

3.5 查看运行时间

xianzhikrylov_print_info() 输出最近一次求解的迭代次数、残差和时间。analysis_seconds、numeric_prepare_seconds、solve_seconds 分别对应三个计算阶段，单位为秒。prepare_solve_seconds 为本次实际执行的准备与求解总时间。

文件读取、对象创建、CPU 与 GPU 间的数据复制、打印和销毁不计入这项时间。分阶段调用时，两次调用之间应用等待的时间也不计入。只重复 SOLVE 时，本次分析和数值准备时间为零。多右端项的求解时间包含所有右端项。

调用外部 AMG 时，amg_setup_seconds 表示最近一次 AMG 准备耗时，amg_apply_seconds 和 amg_apply_calls 表示本次求解中 AMG 的耗时和调用次数。这些是总时间中的分项，用于了解耗时分布。

4 数据类型与存储格式

本章说明如何把自己的矩阵 A 和向量 b 交给库。头文件 XianzhiKrylovData.h 定义数据类型，XianzhiKrylovAPI.h 定义创建和求解函数。数组索引从 0 开始。

4.1 精度与 C 数据类型

类型常量	C 数组元素	含义
XIANZHIKRYLOV_FP64	double	双精度实数。
XIANZHIKRYLOV_FP32	float	单精度实数。
XIANZHIKRYLOV_COMPLEX_FP64	xianzhikrylov_complex_t	real、imag 各为一个 double。
XIANZHIKRYLOV_COMPLEX_FP32	xianzhikrylov_complex_fp32_t	real、imag 各为一个 float。
XIANZHIKRYLOV_DD	高位、低位两个 double 数组	两部分之和表示一个高精度实数。

DD 即 Double-Double，有效精度约为 106 个二进制位。各精度的单右端项样例见第 A 章。

4.2 填写矩阵描述

xianzhikrylov_matrix_data_t 记录矩阵格式、数值类型、尺寸、索引和数值数组。每个数组由 data（数组地址）和 bytes（总字节数）描述。初始化时先清零，再按头文件填写结构版本和大小：

```
xianzhikrylov_matrix_data_t data = {0};
data.abi_version = XIANZHIKRYLOV_ABI_VERSION;
data.struct_size = sizeof(data);
```

保留上述初始化写法即可。按所用矩阵格式填写相关字段，其余数组字段保持零值。

4.3 CSR：按行存储非零元素

CSR 使用三个数组：

- row_offsets：int32_t，长度 $n+1$，记录每行在非零元数组中的起止位置。首项为 0，末项为非零元数 nnz。
- column_indices：int32_t，长度 nnz，记录每个非零元的列号。每行的列号严格递增。
- values：长度 nnz，保存与列号一一对应的矩阵数值。每行包含对角项。

例如：

$$A=\begin{bmatrix}4 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 3\end{bmatrix},\qquad b=\begin{bmatrix}3\\2\\2\end{bmatrix}.$$

下列片段创建这个矩阵和两个向量。初值为零，解应接近全一向量。

```
int32_t row_offsets[] = {0, 2, 5, 7};
int32_t column_indices[] = {0, 1, 0, 1, 2, 1, 2};
double values[] = {4, -1, -1, 4, -1, -1, 3};
```

```
double b[] = {3, 2, 2};
double x[] = {0, 0, 0};

xianzhikrylov_matrix_data_t data = {0};
data.abi_version = XIANZHIKRYLOV_ABI_VERSION;
data.struct_size = sizeof(data);
data.format = XIANZHIKRYLOV_CSR;
data.value_type = XIANZHIKRYLOV_FP64;
data.index_type = XIANZHIKRYLOV_INDEX_I32;
data.block_size = 1;
data.block_layout = XIANZHIKRYLOV_BLOCK_ROW_MAJOR;
data.memory_space = XIANZHIKRYLOV_HOST;
data.device_id = -1;
data.rows = 3;
data.columns = 3;
data.entries = 7;
data.row_offsets.data = row_offsets;
data.row_offsets.bytes = sizeof(row_offsets);
data.column_indices.data = column_indices;
data.column_indices.bytes = sizeof(column_indices);
data.values.data = values;
data.values.bytes = sizeof(values);

xianzhikrylov_matrix_t matrix = NULL;
xianzhikrylov_vector_t rhs = NULL, solution = NULL;
xianzhikrylov_status_t status;

status = xianzhikrylov_matrix_create(&matrix, &data);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
status = xianzhikrylov_vector_create(
    &rhs, 3, b, XIANZHIKRYLOV_FP64, XIANZHIKRYLOV_HOST);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
status = xianzhikrylov_vector_create(
    &solution, 3, x, XIANZHIKRYLOV_FP64, XIANZHIKRYLOV_HOST);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
```

这是数据创建片段，cleanup 表示应用的资源释放位置。完整求解还需创建计算对象、加载配置、调用 execute() 并释放对象，见第 6 章的随包程序。上例数组在 CPU 内存中，因此矩阵描述中的 device_id 填 -1；计算使用的 GPU 编号由 xianzhikrylov_create() 指定。

4.4 其他稀疏格式

4.4.1 BSR：按稠密块存储

BSR 将若干相邻元素组成小方块。rows、columns 和 entries 分别为块行数、块列数和非零块数。块大小填在 block_size；例如 BSR4 的每个块为 4×4。块行指针和块列索引使用 uint64_t 数组，类型常量填写 XIANZHIKRYLOV_INDEX_I64。diagonal_positions 记录每个块行的对角块在块数组中的位置。块数值逐块排列，每个块内部逐行存储，填写 XIANZHIKRYLOV_BLOCK_ROW_MAJOR。向量长度为块行数乘块大小。可参考样例 26_bsr4_fgmres_ilu0。

4.4.2 MSR：对角与非对角分开存储

MSR 使用 values 保存对角值，offdiagonal_values 保存非对角值。row_offsets 和 column_indices 为 int32_t 数组，描述非对角连接，entries 是非对角连接数。

块大小为 s 时，rows、columns 按块计数，每个对角块有 s^2 个元素，块内逐行存放。每个非对角连接保存一个标量，表示该标量乘 $s \times s$ 单位阵。$s = 1$ 是标量 MSR，$s = 3$ 是此类三分量系统。标量示例见 29_msr_bicgstab_jacobi。

4.4.3 LDU：对角、下三角与上三角分开存储

values 保存对角值。每个连接的两个行号记为 $i < j$：lower_address 保存 i，upper_address 保存 j；lower_values 保存 A_{ji}，upper_values 保存 A_{ij}。地址使用 int32_t，entries 为连接数，rows 为标量行数。这种写法在有限体积计算中常用；示例见 30_ldu_pcg_dic 和 31_ldu_bicgstab_dilu。

4.5 一个向量与多个向量

一个连续向量使用 xianzhikrylov_vector_create()。对于多个右端项，使用 xianzhikrylov_multiple_vector_create()，并指定每个向量长度 length、向量个数 rhs_count、数组步长 leading_dimension 和排列方式：

排列方式	第 j 个向量的第 i 个元素
XIANZHIKRYLOV_RHS_MAJOR	一个向量存完再存下一个：data[j*leading_dimension+i]，步长至少为 length。
XIANZHIKRYLOV_ROW_MAJOR	同一行的多个向量元素放在一起：data[i*leading_dimension+j]，步长至少为 rhs_count。

步长按元素个数计算。普通方法可传入 1–10 个右端项，外部 AMG 方法一次处理一个。矩阵、b、x 的标量行数应一致；b 和 x 的向量个数应一致。

4.6 DD 输入

DD 的矩阵输入采用 CSR，values 与 values_low 分别保存高位与低位数组。向量同样提供两个数组，数学值为两部分之和。创建单个 DD 向量时，使用 xianzhikrylov_multiple_vector_create()，将 rhs_count 设为 1，data 指向高位数组，low 指向低位数组。这里仍然只有一个右端项；这个函数用于同时传入两个数值数组。可运行示例为 35_dd_fgmres。

5 函数参考

公开接口包含十个函数，所有函数返回 xianzhikrylov_status_t。XIANZHIKRYLOV_SUCCESS 表示操作完成；XIANZHIKRYLOV_CONVERGED 表示求解阶段满足停止条件。

函数	用途
xianzhikrylov_create()	创建绑定设备的计算句柄。
xianzhikrylov_config_set()	加载求解 JSON 和授权配置。
xianzhikrylov_matrix_create()	创建矩阵描述对象。
xianzhikrylov_vector_create()	创建连续单 RHS 向量。
xianzhikrylov_multiple_vector_create()	创建多 RHS、显式布局或 DD 向量。
xianzhikrylov_execute()	执行分析、数值准备或求解阶段。
xianzhikrylov_print_info()	打印最近一次求解统计。
xianzhikrylov_matrix_destroy()	销毁矩阵对象。
xianzhikrylov_vector_destroy()	销毁向量对象。
xianzhikrylov_destroy()	销毁计算句柄。

5.1 xianzhikrylov_create()

```
xianzhikrylov_status_t xianzhikrylov_create(
    int32_t device_id,
    xianzhikrylov_handle_t *handle);
```

参数	方向	说明
device_id	输入	当前进程可见设备编号，通常为 0。
handle	输出	成功时接收计算句柄；失败时置为 NULL。

5.2 xianzhikrylov_config_set()

```
xianzhikrylov_status_t xianzhikrylov_config_set(
    xianzhikrylov_handle_t handle,
    xianzhikrylov_config_option_t option,
    const char *value);
```

参数	方向	说明
handle	输入	已成功创建的计算句柄。
option	输入	填写 XIANZHIKRYLOV_CONFIG_FILE。
value	输入	以空字符结尾的 JSON 文件路径。

函数读取 JSON、选择方法／预条件器并完成配置阶段授权检查。必须在首次执行依赖配置的 execute() 之前调用。库复制路径字符串，不保留调用方指针。

5.3 xianzhikrylov_matrix_create()

```
xianzhikrylov_status_t xianzhikrylov_matrix_create(
    xianzhikrylov_matrix_t *matrix,
    const xianzhikrylov_matrix_data_t *data);
```

参数	方向	说明
matrix	输出	成功时接收矩阵对象。
data	输入	矩阵格式、数值类型、尺寸和数组等信息，见第 4 章。

该函数只创建描述对象，不执行完整设备分配、结构分析或预条件器构建。矩阵数组由调用方管理。

5.4 xianzhikrylov_vector_create()

```
xianzhikrylov_status_t xianzhikrylov_vector_create(
    xianzhikrylov_vector_t *vector,
    int64_t size,
    void *data,
    xianzhikrylov_value_type_t value_type,
    xianzhikrylov_memory_space_t memory_space);
```

参数	方向	说明
vector	输出	成功时接收向量对象。
size	输入	元素个数，应等于矩阵阶数。
data	输入/输出	连续数据缓冲区；用于解对象时由库写入结果。
value_type	输入	向量的数值类型。
memory_space	输入	HOST 或 DEVICE。

单个向量的元素连续存储。DD 向量使用 multiple_vector_create 同时传入高位与低位数组，rhs_count 填 1。

5.5 xianzhikrylov_multiple_vector_create()

```
xianzhikrylov_status_t xianzhikrylov_multiple_vector_create(
    xianzhikrylov_vector_t *vector,
    int64_t length,
    int64_t rhs_count,
    int64_t leading_dimension,
    xianzhikrylov_layout_t layout,
    void *data,
    void *low,
    xianzhikrylov_value_type_t value_type,
    xianzhikrylov_memory_space_t memory_space);
```

参数	方向	说明
vector	输出	成功时接收向量对象。
length	输入	每个 RHS 的元素数。
rhs_count	输入	普通方法为 1–10；外部 AMG 方法为 1。
leading_dimension	输入	列间或行间跨度，按完整数值元素计数。
layout	输入	XIANZHIKRYLOV_RHS_MAJOR 或 XIANZHIKRYLOV_ROW_MAJOR。
data	输入/输出	数据数组；DD 时为高位数组。
low	输入/输出	DD 的低位数组；其他类型填 NULL。
value_type	输入	数值类型。
memory_space	输入	HOST 或 DEVICE。

5.6 xianzhikrylov_execute()

```
xianzhikrylov_status_t xianzhikrylov_execute(
    xianzhikrylov_handle_t handle,
    xianzhikrylov_phase_t phase,
    xianzhikrylov_matrix_t matrix,
    xianzhikrylov_vector_t solution,
    xianzhikrylov_vector_t rhs);
```

参数	方向	说明
handle	输入	已成功配置的计算句柄。
phase	输入	分析、数值准备、求解或完整执行阶段。
matrix	输入	矩阵 A 对象。
solution	输入/输出	分析／数值准备阶段为 NULL；求解阶段保存初值及最终解 x。
rhs	输入	分析／数值准备阶段为 NULL；求解阶段为右端项对象。

阶段	参数与结果
PHASE_ANALYSIS	只要求 matrix；返回 SUCCESS。
PHASE_NUMERIC_PREPARE	只要求 matrix；返回 SUCCESS。
PHASE_SOLVE	要求 A、x、b；全部 RHS 收敛时返回 CONVERGED。
PHASE_ALL	自动完成准备和求解；全部 RHS 收敛时返回 CONVERGED。

5.7 xianzhikrylov_print_info()

```
xianzhikrylov_status_t xianzhikrylov_print_info(
    xianzhikrylov_handle_t handle);
```

打印最近一次求解的状态、每 RHS 迭代次数、方法编号、残差、右端项范数和准备／求解时间。该函数不重新求解。handle 为已完成求解的计算对象。打印成功返回 SUCCESS。

5.8 xianzhikrylov_matrix_destroy()

```
xianzhikrylov_status_t xianzhikrylov_matrix_destroy(
    xianzhikrylov_matrix_t matrix);
```

matrix 为要销毁的矩阵对象。销毁矩阵描述对象，不释放应用提供的数组。成功后应用应将保存对象的变量置为 NULL。

5.9 xianzhikrylov_vector_destroy()

```
xianzhikrylov_status_t xianzhikrylov_vector_destroy(
    xianzhikrylov_vector_t vector);
```

vector 为要销毁的向量对象。该函数同时适用于单个和多个向量；数组仍由应用释放。调用成功后，由应用将保存对象的变量置为 NULL。

5.10 xianzhikrylov_destroy()

```
xianzhikrylov_status_t xianzhikrylov_destroy(
    xianzhikrylov_handle_t handle);
```

handle 为要销毁的计算对象。函数释放配置、计算工作区、准备结果和外部 AMG 对象。调用成功后，由应用执行 handle=NULL;。

5.11 分阶段执行与连续求解

ANALYSIS 只建立结构信息；NUMERIC_PREPARE 读取当前矩阵系数并完成数值准备；SOLVE 读取右端和初值并求解。CG 加 Jacobi 的数值检查发生在数值准备阶段，分析成功不表示矩阵数值满足方法要求。每一阶段都必须检查返回值。

- 仅更换右端或初值且缓冲区地址不变时，更新对应数组后再次调用SOLVE。初值是否使用由initial_guess 控制。
- 结构与缓冲区地址不变、系数改变时，修改系数数组后调用NUMERIC_PREPARE，再调用SOLVE；无需重复分析。
- 结构内容改变而描述信息和缓冲区地址仍有效时，重新调用ANALYSIS、NUMERIC_PREPARE、SOLVE。
- 若尺寸、数组地址或布局改变，应重新创建相应描述对象；新矩阵对象须重新分析及数值准备。

本接口通过执行对应阶段通知库更新数据，没有单独的公开matrix_update 函数。不能修改系数后只调用SOLVE 并期待预条件器自动更新。Samples 中的37_cg_jacobi_staged 演示右端、系数及结构更新，并逐次独立验算；36_multiple_rhs_reuse 演示多右端复用。

6 样例与求解配置

6.1 从随包样例开始

17_fgmres_ilu0 使用 FGMRES 迭代法和 ILU(0) 预条件器，求解一个百万阶、一般非对称的 CSR 系统。按第 2 章设置好 XK_ROOT 后，在同一个终端中运行：

```
cd "$SAMPLES_ROOT/17_fgmres_ilu0"
make LIBRARY_ROOT="$XK_ROOT"
make run LIBRARY_ROOT="$XK_ROOT"
```

本目录的 sample.c 是完整程序，使用 common/sample_common.h 中的辅助函数：

辅助函数	作用
xk_matrix_load	读取矩阵结构，生成数值和 b。
xk_binding_make	填写矩阵描述，调用矩阵创建函数和 b 的向量创建函数。
xk_solution_init	分配清零的 x 数组，创建解向量对象。
xk_verify	用原始 A、b 和输出 x 独立计算残差。
xk_*_free	销毁数据对象，释放样例分配的数组。

这些辅助函数属于样例代码，便于阅读和复用。下面列出与仓库一致的 sample.c；它与上述公共头文件和矩阵文件一起编译运行。

```
/* XianzhiKrylov sample 17_fgmres_ilu0: fgmres / ilu0. */
#include "sample_common.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    const char *matrix_path = argc > 1 ? argv[1] : "../matrices/transport_1m.csrpat";
    const char *json_path = argc > 2 ? argv[2] : "config.json";
    xk_matrix matrix = {0};
    xk_binding binding = {0};
    xk_solution solution = {0};
    xianzhikrylov_handle_t handle = NULL;
    xianzhikrylov_status_t status;
    int rc = 2, destroyed = 0;

    if (!xk_matrix_load(&matrix, matrix_path))
        goto done;
    if (!xk_binding_make(&binding, &matrix, XIANZHIKRYLOV_CSR, 1)) {
        fprintf(stderr, "cannot prepare matrix format\n");
        goto done;
    }
    if (!xk_solution_init(&solution, &matrix)) {
        fprintf(stderr, "cannot allocate solution\n");
        goto done;
    }

    status = xianzhikrylov_create(0, &handle);
    if (status != XIANZHIKRYLOV_SUCCESS) {
        fprintf(stderr, "create status=%d\n", (int)status);
        rc = 1;
        goto done;
    }
    status = xianzhikrylov_config_set(
        handle, XIANZHIKRYLOV_CONFIG_FILE, json_path);
```

```
    if (status != XIANZHIKRYLOV_SUCCESS) {
        fprintf(stderr, "config_set status=%d\n", (int)status);
        rc = 1;
        goto done;
    }
    /* HOST A, b, and x are prepared by the library during execute(ALL). */
    status = xianzhikrylov_execute(
        handle, XIANZHIKRYLOV_PHASE_ALL, binding.matrix_object,
        solution.vector_object, binding.rhs_object);
    if (status != XIANZHIKRYLOV_CONVERGED) {
        fprintf(stderr, "execute status=%d (not converged)\n", (int)status);
        rc = 1;
        goto done;
    }
    {
        const xianzhikrylov_status_t print_status =
            xianzhikrylov_print_info(handle);
        const xianzhikrylov_status_t destroy_status =
            xianzhikrylov_destroy(handle);
        handle = NULL;
        const int residual_ok = xk_verify(&matrix, &solution, 1e-8L);
        destroyed = 1;
        rc = print_status == XIANZHIKRYLOV_SUCCESS &&
             destroy_status == XIANZHIKRYLOV_SUCCESS &&
             residual_ok ? 0 : 1;
    }

done:
    if (handle && !destroyed)
        (void)xianzhikrylov_destroy(handle);
    xk_solution_free(&solution);
    xk_binding_free(&binding);
    xk_matrix_free(&matrix);
    return rc;
}
```

6.2 本样例的配置文件

下面是 17_fgmres_ilu0/config.json 的完整内容：

```
{
  "schema_version": 2,
  "matrix_format": "csr",
  "block_size": 1,
  "solver": {
    "method": "fgmres",
    "preconditioner": "ilu0",
    "max_iterations": 2000,
    "restart": 40,
    "relative_tolerance": 1e-09,
    "absolute_tolerance": 0,
    "initial_guess": "zero",
    "level": 0
  }
}
```

字段	如何填写
schema_version	新配置填写 2；已有 schema 1 配置继续兼容。
matrix_format	auto、csr、bsr、msr、ldu；auto 从矩阵对象取得格式。
matrix_symmetry	general 为一般矩阵，spd 为对称正定矩阵。
block_size	标量 CSR 填 1；块矩阵填写实际块大小。
solver.method	迭代法名称，见下表。
solver.preconditioner	预条件器名称，见下表。
solver.level	所选求解器与预条件器内部的可选档位，见第6.4节。
solver.max_iterations	最大迭代次数；达到上限时程序返回对应状态。
solver.restart	GMRES、FGMRES、GCR 每轮保留的迭代向量数；较大会增加内存用量。
solver.relative_tolerance	迭代中的相对停止容差，省略时为1e-6；数值越小，要求越严格。
solver.absolute_tolerance	绝对残差阈值；与相对阈值的关系见第 7 章。
solver.initial_guess	zero、provided、reuse，见本章初值说明。

6.3 选择方法和预条件器

先根据矩阵性质选择随包样例，再在它的完整配置上调整容差和迭代次数。CG、PCG 和 FCG 适用于对称正定矩阵；IC(0) 同样要求适合不完全 Cholesky 分解的对称正定矩阵。一般非对称矩阵可从 FGMRES + ILU(0) 开始。

格式	预条件器配置值	迭代法配置值
CSR	none（或 identity）	cg, bicg, bicgstab, gmres, fgmres, gcr。
CSR	jacobi	pcg, fcg, bicg, bicgstab, gmres, fgmres, gcr。
CSR	ilu0	bicg, bicgstab, gmres, fgmres, gcr。
CSR	ic0	pcg, fcg, bicg, bicgstab, gmres, fgmres, gcr。
BSR4	ilu0, block_jacobi, multicolor_block_dilu	fgmres。
BSR3	ilu0	fgmres（L0）。
MSR	jacobi	bicgstab。
LDU	dic	pcg。
LDU	dilu	bicgstab。

上述名称选择数学方法。支持某组合不保证任意矩阵都能在指定上限内收敛；有兼容回退规则的组合会提示实际使用的配置，其余不支持的组合返回对应状态。CSR BiCG 与 ILU0 只支持 L0，默认 L0。

CUDA 原生 BSR 块大小为 2、3、4、5。其他块大小的输入先转换为 CSR，求解按所选路线的 CSR 支持范围处理。展开必须满足本地 CSR 索引和内存容量限制。原生 BSR3 支持 FP64 的 FGMRES 与块 ILU(0)，使用零初值，默认档为 L0。JSON v2 显式请求 L1 或 L2 时，库提示一次并以 L0 继续求解。

使用 CUDA 26.44 或 26.45 时，请使用 JSON v1。独立 Samples 提供转换工具，可将目标版本能等价表达的 FP64 JSON v2 配置转为 JSON v1。

6.4 方法、预条件器与 Level

在 schema 2 中先填写`solver.method` 及`solver.preconditioner`，再按需要设置`solver.level`。ILU0 支持不同 Level：L0 偏重预条件精度，L1 偏重速度。各档均使用配置的收敛容差。省略 Level 时使用该组合的默认档。

`parilu0` 是独立预条件器，因子扫描数通过`parameters.factor_sweeps` 指定。其他预条件器只接受已登记档位，通常为 L0。按该组合支持的档位填写 Level。

schema 1 文件继续采用原有名称和 Level 快捷语义，不重新解释旧 Level 编号。迁移为 schema 2 时，先明确原求解器和预条件器，再选择相应 Level。以下为完整 schema 2 配置：

```
{
  "schema_version": 2,
  "matrix_format": "csr",
  "block_size": 1,
  "solver": {
    "method": "bicgstab",
    "preconditioner": "ilu0",
    "level": 0,
    "max_iterations": 1000,
    "initial_guess": "zero"
  }
}
```

本例使用 BiCGStab 和 ILU0 L0；省略的相对容差为`1e-6`。schema 1、schema 2 和直接 C 配置初始化的默认相对容差统一为`1e-6`，显式设置保持不变。内部实现由库自动选择，用户无需配置性能策略。

6.5 详细预条件器与参数

以下组合也可通过相同的 JSON 入口选择。`method_options` 与`parameters` 是顶层对象，与`solver` 并列。

格式	选择	说明
CSR	GMRES + `block_jacobi`	`method_options.csr_jacobi_block_size` 选择 2、3 或 4；`cgs_passes` 选择 1 或 2。输入描述符的块大小仍为 1。
CSR	CG + `ssor`	`method_options.ssor_omega` 在 0 与 2 之间；取 1 表示对称 Gauss–Seidel。矩阵需声明 SPD。
CSR	FGMRES + `csr_block_jacobi_ilu0` 或`csr_partitioned_ilu0`	`parameters.partitions` 指定非空分区数，范围 1 到标量行数；保留输入顺序。
CSR	`parilu0`	支持 FGMRES、GMRES、BiCGStab、GCR；`parameters.factor_sweeps` 在 1 到 64 之间。
CSR	`csr_multicolor_dilu`	支持 FGMRES、GMRES、BiCGStab、GCR。
BSR2/4/5	FGMRES + `parilu0`、`block_jacobi`、`multicolor_block_dilu`	原生块预条件方法。`block_jacobi_ilu0` 与`partitioned_ilu0` 还接收`parameters.partitions`。
BSR2/4/5	FGMRES + `identity`、`csr_block_jacobi_ilu0` 或`csr_partitioned_ilu0`	明确选择标量方法；保留存储的全部系数和标量顺序，转换计入执行时间。不可与原生块 ILU 混用。

MSR1/3	jacobi 或block_jacobi	支持 CG、FCG、BiCGStab、GCR；MSR3 的 Jacobi 逐分量缩放，块 Jacobi 使用 3 乘 3 块逆。CG 与 FCG 要求 SPD。
MSR1/3	GCR + hsgs	当前支持的对称 Gauss–Seidel 预条件路线；其他 HSGS 变体不在本表支持范围。
LDU	CG + jacobi 或dic	要求 SPD；DIC 不满足数值条件时返回相应失败状态。
LDU	FGMRES、BiCG、BiCGStab + jacobi 或dilu	DILU 与 ILU0 是不同的预条件器。

例如，GMRES 加 2 乘 2 块 Jacobi 并使用两遍 CGS：

```
{
  "schema_version": 2,
  "matrix_format": "csr",
  "block_size": 1,
  "solver": {
    "method": "gmres",
    "preconditioner": "block_jacobi",
    "restart": 32,
    "max_iterations": 1000
  },
  "method_options": {
    "cgs_passes": 2,
    "csr_jacobi_block_size": 2
  }
}
```

未知字段、冲突字段或未登记的组合会返回错误。Level 不代替这些数学参数，库不会为了收敛而更换用户选定的方法。

6.6 分阶段调用与数据更新

以下为调用片段；每一步都应检查返回状态，成功后再执行下一步：

```
status = xianzhikrylov_execute(
    handle, XIANZHIKRYLOV_PHASE_ANALYSIS,
    matrix, NULL, NULL);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
status = xianzhikrylov_execute(
    handle, XIANZHIKRYLOV_PHASE_NUMERIC_PREPARE,
    matrix, NULL, NULL);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
status = xianzhikrylov_execute(
    handle, XIANZHIKRYLOV_PHASE_SOLVE,
    matrix, solution, rhs);
if (status != XIANZHIKRYLOV_CONVERGED) goto cleanup;
```

只改变 b 时，更新它的数组并再次执行 SOLVE。A 的结构不变、仅数值改变时，先重做 NUMERIC_PREPARE，再执行 SOLVE。每次显式调用ANALYSIS 都会重建结构分析并使数值准备失效；ALL 包含完整分析、准备和求解。只有明确只更新值或右端时才复用已有准备。完整状态检查和复用示例见 36_multiple_rhs_reuse/sample.c。

6.7 初值

默认可在完整配置的 solver 对象中设置 "initial_guess":"zero"，表示从零向量开始。若已有近似解，将它写入 x 数组，并将该字段改为 provided。reuse 也读取调用方当前的 x，保留上次求解结果即可用作下次初值。

本节说明使用 solver 配置对象的 CSR 方法。精度样例使用各自的 extended_solver 配置，按随包文件中的零初值运行。BSR4 FGMRES + ILU(0) 使用上述 bsr_jacobi4 配置时可读取非零初值；其他随包 BSR 样例和外部 AMG 方法使用零初值。

6.8 分别学习各精度

32_complex_fgmres、33_float_fgmres_jacobi、34_float_complex_fgmres 和 35_dd_fgmres 均只求解一个右端项。它们分别展示复数双精度、实数单精度、复数单精度和 DD 的 A、b、x。使用各目录中的完整 config.json，其中 extended_solver.method 是相应方法的配置名称。复数配置名中的 batch 是函数库使用的名称；第 32 个样例将 batch 设为 1。

6.9 多右端项：double 样例

36_multiple_rhs_reuse 使用 double 的 A、b、x，三个右端项共用同一矩阵和预条件器。例如，将每个长度为 n 的向量连续存放，三个向量依次排在数组中：

```
status = xianzhikrylov_multiple_vector_create(
    &rhs, n, 3, n, XIANZHIKRYLOV_RHS_MAJOR,
    b, NULL, XIANZHIKRYLOV_FP64, XIANZHIKRYLOV_HOST);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
status = xianzhikrylov_multiple_vector_create(
    &solution, n, 3, n, XIANZHIKRYLOV_RHS_MAJOR,
    x, NULL, XIANZHIKRYLOV_FP64, XIANZHIKRYLOV_HOST);
if (status != XIANZHIKRYLOV_SUCCESS) goto cleanup;
```

这里每个数组包含 $3n$ 个 double。分别检查两个创建函数的返回状态，再调用 execute()。样例还更新 b 后重复求解，并逐个检查结果。

6.10 与 LinglongAMG 配合

AMG（代数多重网格）可以作为外部预条件器：XianzhiKrylov 负责外层迭代，LinglongAMG 每次执行一个 AMG 周期来改善当前修正量。两库分别管理自己的计算对象，应用通过 XianzhiKrylov 接口传入 A、b、x。

NVIDIA 和海光软件包中，DD-FGMRES 外层可使用以下完整配置。请将两条路径替换为实际的 LinglongAMG 共享库和配置文件路径：

```
{
  "extended_solver": {
    "method": "fgmres_dd",
    "provider": "linglongamg",
    "library_path": "/path/to/liblinglongamg.so",
    "config_path": "/path/to/amg_config.json",
    "restart": 15,
    "max_cycles": 10,
    "relative_tolerance": 1e-9,
    "inner_method": "none"
  }
}
```

该组合输入 FP64 的 A、b，输出 DD 的 x，一次求解一个右端项，从零初值开始。LinglongAMG 的配置选择一次完整周期，例如 `method=amg`、`max_iters=1`；外层收敛要求由 XianzhiKrylov 的 `relative_tolerance` 控制。

另一种配置名为 `projected_pcg`，A、b、x 均为 FP64，通过 `max_iterations` 设置迭代上限，用于适合常数模态投影 PCG 的系统。普通对称正定系统可使用前面的 PCG 样例。

AMG 准备发生在 `NUMERIC_PREPARE`，每次 AMG 调用发生在 `SOLVE`。两库使用各自的安装文件和授权；LinglongAMG 的安装与其配置字段见对应产品手册。

7 输出、验证与常见问题

7.1 如何判断求解成功

创建对象、加载配置等操作成功时返回 SUCCESS（0）。求解阶段返回 CONVERGED（1），表示所有右端项均满足库的停止条件。样例还使用原始 A 和 b 计算残差，检查输出的 x，通过后打印 PASS。

输出字段	含义
iterations	实际迭代次数。多右端项时逐个输出统计。
converged	1 表示满足停止条件。
method	本次使用的方法编号。方法由配置文件选择。
residual	所选方法采用的残差度量。
rhs_norm	向量 b 的范数，即它的整体大小。
relative_residual	残差与 b 的范数之比。
prepare_solve_seconds	准备与求解总时间，单位为秒。
relative_l2、absmax	样例独立计算的 L2 相对残差与最大分量绝对残差。

7.2 残差与容差

残差为 $r = b - Ax$。它越小，计算出的 x 越接近满足原方程。L2 范数为 $\|r\|_2 = \sqrt{\sum_i |r_i|^2}$。普通 FP64 方法的迭代停止判据为：

$$\|r_{\text{rec}}\|_2 \le \max\bigl(atol,\ rtol\,\|b\|_2\bigr).$$

relative_tolerance 对应 $rtol$，absolute_tolerance 对应 $atol$。两个阈值取较大者，所以达到其中一个阈值即可。相对残差为 $\|r\|_2/\|b\|_2$；$b = 0$ 时使用绝对残差。

FP32、复数 FP32 和 DD 输入方法的 tolerance 表示相对 L2 容差，absolute_tolerance 表示绝对 L2 容差。外部 AMG 方法使用 relative_tolerance 控制外层 L2 相对残差。

复数 FP64 方法 fgmres_complex_batch 的 tolerance 表示 $\max_i |r_i|$ 的绝对阈值；复数的 $|r_i|$ 是其模。该方法输出residual_linf、rhs_norm_same_norm 和 relative_residual_same_norm，分别表示绝对最大残差、右端最大范数及两者之比。Samples 另行计算原系统的相对 L2 残差。

7.3 可选真实残差验收

常规 FP64 配置省略solver.relative_tolerance 时，默认相对停止容差为1e-6；绝对容差默认是 0。迭代过程中采用递推残差或 GMRES 类方法的残差估计判断停止。结束后，库用原始矩阵、右端项和实际输出解重新计算真实残差，并保存统计；打印信息不会重新求解。

可选布尔字段solver.verify_true_residual 默认是false，Samples 配置不写这个字段。关闭时，CONVERGED 表示迭代停止判据已满足，末尾真实残差仅用于报告。设为true 后，库还要求真实残差满足同一容差：首次检查未通过时，从当前解续算至多 10 个迭代步，再次检查；此后每 10 步检查一次，直到通过或达到max_iterations。最后剩余步数少于 10 时，在总迭代上限处检查，不增加或重置迭代预算。数值或设备错误仍按对应错误码结束。

真实残差通过时返回CONVERGED；到上限仍未通过时返回MAX_ITERATIONS，保留最终候选解和残差统计。多右端项逐个验收，全部通过才返回收敛。额外检查和续算的耗时包含在求解时间中。极小容差不自动放宽；递推残差很小而真实残差停在舍入误差附近时，开启验收可能一直续算到上限。显式设置的容差保持不变。

7.4 独立验证

用传入的原始 A、b 和得到的 x 重新计算 $b - Ax$，再与所需容差比较。样例的验证代码位于 common/sample_common.h 或各精度的 sample.c 中。DD 验证应同时使用高位与低位数组；需要严格高精度验证时，使用足够精度的独立计算。API 操作成功和解达到数值要求是两件事，应用应同时检查。

7.5 返回状态

值	常量	含义与处理
0	XIANZHIKRYLOV_SUCCESS	操作完成。
1	XIANZHIKRYLOV_CONVERGED	所有右端项达到停止条件。
2	XIANZHIKRYLOV_MAX_ITERATIONS	达到迭代上限；检查容差、方法和预条件器。
3	XIANZHIKRYLOV_BREAKDOWN	迭代中出现无法继续计算的情况；查看原始诊断和矩阵性质。
4	XIANZHIKRYLOV_INVALID_ARGUMENT	检查参数、尺寸、数组字节数及初始化字段。
5	XIANZHIKRYLOV_UNSUPPORTED	按该软件包的样例选择格式、精度、方法和内存位置的组合。
6	XIANZHIKRYLOV_STALE_SYSTEM	数据已更新；按第 3 章重新执行所需准备阶段。
7	XIANZHIKRYLOV_DEVICE_ERROR	检查 GPU 状态和工具包环境。
8	XIANZHIKRYLOV_STATE_ERROR	检查创建、配置、准备、求解的调用顺序。
9	XIANZHIKRYLOV_RESIDUAL_MISMATCH	检查原系统残差和所设容差。
10	XIANZHIKRYLOV_STAGNATED	残差停止下降；检查方法和预条件器是否适合矩阵。
11	XIANZHIKRYLOV_LICENSE_ERROR	检查授权文件、设备、有效期及软件包。

调用失败后，先处理该错误，再继续依赖它的步骤。返回 2、3、9、10 或 11 时应保留诊断信息，按失败处理。

NVIDIA 软件包的 CG/PCG 求解返回BREAKDOWN 时，可调用xianzhikrylov_print_info() 查看已完成的迭代次数和残差信息。这些统计用于判断中断原因，不代表解已达到所设容差。

7.6 常见问题

现象	下一步
找不到 XianzhiKrylovAPI.h	检查 LIBRARY_ROOT 是否指向含有 include/ 和 lib/ 的包目录。
运行时缺少共享库	执行 ldd ./sample；按第 2 章填写工具包运行库目录。
config_set 返回 4	确认配置文件路径存在，JSON 括号与逗号完整。
配置或求解返回 11	检查授权位置、有效期及授权对应的 GPU，见第2.4节。
matrix_create 返回 4	按第 4 章检查尺寸、索引类型、数组长度和字节数。
向量创建返回 4	检查向量长度和数值类型；多向量还需检查个数、步长和排列方式。
求解未收敛	检查 A、b、初值、容差、迭代上限及方法适用性。
修改代码后仍运行旧程序	在样例目录执行 makeclean，再执行 make 和 makerun。

附录 A　样例清单与矩阵文件

A.1　按求解方法选择样例

每个样例目录包含 `sample.c`、`config.json`、`Makefile` 和 `README.md`。以下常用方法样例均使用 FP64（`double`）：

目录编号	方法	矩阵格式
01–06	CG、BiCG、BiCGStab、GMRES、FGMRES、GCR	CSR，无预条件器。
07–13	PCG、FCG、BiCG、BiCGStab、GMRES、FGMRES、GCR + Jacobi	CSR。
14–18	BiCG、BiCGStab、GMRES、FGMRES、GCR + ILU(0)	CSR。
19–25	PCG、FCG、GMRES、FGMRES、BiCGStab、BiCG、GCR + IC(0)	CSR，对称正定矩阵。
26–28	FGMRES + ILU(0)、块 Jacobi、多色块 DILU	BSR4。
29	BiCGStab + Jacobi	MSR。
30–31	PCG + DIC、BiCGStab + DILU	LDU。

完整目录名及每个样例的运行命令见 `xianzhikrylov_samples/README.md`。

A.2　精度样例：每个只有一个右端项

以下样例在 NVIDIA 和海光软件包中使用：

目录	A、b、x 的精度
`17_fgmres_ilu0`	FP64：实数 `double`。
`32_complex_fgmres`	复数 FP64：实部、虚部各为 `double`。
`33_float_fgmres_jacobi`	FP32：实数 `float`。
`34_float_complex_fgmres`	复数 FP32：实部、虚部各为 `float`。
`35_dd_fgmres`	DD：高位、低位两组 `double` 数组。

DD 样例使用 `multiple_vector_create` 传入高、低位数组，其中 `rhs_count=1`，仍然求解一个系统。

A.3　多右端项：单独的 double 样例

`36_multiple_rhs_reuse` 使用同一 CSR 矩阵求解三个 FP64 右端项。先分阶段分析、准备、求解，再改变 b，复用准备结果求解。每个解都独立检查残差。

A.4 如何编译

进入任一个样例目录，执行：

```
make
make run
make clean
```

在 Samples 根目录运行 make 可以编译全部样例。保留 common/、matrices/ 和公共构建文件与各样例的相对位置。

A.5 矩阵文件与数值生成

为了减小下载体积，.csrpat 文件保存 CSR 的稀疏结构和固定生成参数。程序读取后生成矩阵数值和 b。相同文件生成相同数据。

文件	矩阵阶数	特点
diffusion_1m.csrpat	1,000,000	对称正定、严格对角占优。
transport_1m.csrpat	1,000,000	一般非对称、严格对角占优。
coupled4_500k.csrpat	500,000	对称块结构、严格对角占优。
complex_262k.csrpat	262,144	复数、严格对角占优，一个右端项。

普通单右端项样例以全一向量构造 $b = A\mathbf{1}$，求解仍从配置指定的初值开始。复数全一向量的每个元素为 $1 + 0i$。单精度样例将生成的数据存入对应的 float 数组，并按这些数组验证残差。DD 样例为矩阵、右端项和解提供高位、低位数组。多右端项样例使用 $A\mathbf{1}$ 及其两倍、三倍作为三个 b，第二轮将它们各减半。

A.5.1 文件结构

XKPAT1 文件采用小端字节序，64 字节文件头后依次为 int32_trow_offsets[n+1] 和 int32_tcolumn_indices[nnz]。文件头记录格式版本、实数或复数模型、阶数、非零元数、右端项数、建议块大小、模型编号和固定种子。完整字节偏移与生成规则见 matrices/PATTERN_FORMAT.md。

接入自己的矩阵时，直接按第 4 章填写数组和矩阵描述即可。

附录 B　文档与技术支持

B.1　在哪里查找说明

文件或位置	内容
包根目录 README.md	本平台软件包的安装和运行要求。
include/XianzhiKrylovAPI.h	C 函数声明与返回状态。
include/XianzhiKrylovData.h	数据类型、矩阵格式与数组描述。
xianzhikrylov_samples/README.md	样例导航和编译运行步骤。
样例目录 README.md	本样例的用途、输入和结果检查。
Samples 的 docs/	中文、英文产品手册与常见问题。
THIRD_PARTY_NOTICES.txt	第三方组件许可与归属说明。

B.2　软件包与授权

可从以下网站下载软件包与授权文件：

https://store.solved-tech.com/

B.3　联系技术支持

请求帮助时，提供软件包文件名、GPU 型号、完整错误输出、配置文件，以及出问题的样例名称或简化程序，便于定位。矩阵规模、精度和独立残差也有助于判断求解问题。授权文件和业务数据请通过约定的私密渠道传送。

授权获取网页为 https://store.solved-tech.com/，二维码见末页。

附录 C　版本日志

本章按从新到旧列出产品的主要更新。

XianzhiKrylov 26.48

针对 NVIDIA GPU 有如下改动。

新增特性：

- 新增原生 BSR3 求解路线。`block_size` 为 3 时，支持 FP64 的 FGMRES ＋块 ILU(0) L0，使用零初值。
- 扩展 BSR 输入兼容性。原生块大小覆盖 2、3、4、5。其他合法块大小在所选求解组合支持 CSR 时，会提示并自动转 CSR 后求解。
- Samples 新增`convert_json_v2_to_v1.py`，可将 JSON v2 配置转换为 XianzhiKrylov 26.44、XianzhiKrylov 26.45 可用的 JSON v1。

改进性能：

- 降低大规模 CSR GMRES ＋ Jacobi 的显存占用，缩短部分大规模系统的求解时间。
- 修复部分矩阵采用 BiCGStab 或 GMRES ＋ ILU(0) L0 时的性能回退。

针对海光 DCU 有如下改动。

新增特性：

- 支持 JSON v2。通过`xianzhikrylov_config_set` 加载`schema_version` 为 2 的配置，可用`solver.method`、`solver.preconditioner` 和`solver.level` 选择求解组合，并用`solver.verify_true_residual` 按需启用原系统真实残差检查。
- Samples 新增`convert_json_v2_to_v1.py`，可将 JSON v2 配置转换为 XianzhiKrylov 26.44、XianzhiKrylov 26.45 可用的 JSON v1。

改进性能：

- 改善混合精度 CG、Double Double FGMRES、LinglongAMG 预条件求解和 3×3 块 Jacobi BiCGStab 的执行效率。
- 缩短复数多右端项 FGMRES ＋ ILU(0) 的预条件准备时间。

XianzhiKrylov 26.46

- 增加 schema 2，按求解器、预条件器和 Level 选择配置，兼容 schema 1 与原 C 接口。
- 统一默认相对容差为`1e-6`；显式容差和可选真实残差验收保持独立。
- 改进分析、数值准备及精确 ILU0 路线的执行效率，完善重复分析与数据更新语义。
- 补充详细配置诊断及矩阵格式、方法、预条件器组合说明。
- 修复 CG 加 Jacobi 分阶段分析误报数值失败的问题，保留旧接口兼容。

- 优化部分 CG、FCG、BiCG、DILU、MSR 及 IC0 路径的执行开销。

XianzhiKrylov 26.45

- 相对停止容差默认改为1e-6，增加可选真实残差验收与每 10 步续算检查。
- 扩展预条件方案的适用迭代方法，增加性能档位设置。
- 提升 NVIDIA 和海光平台的求解性能与稳定性。
- 完善与 PanguLU、LinglongAMG 的组合求解。
- 改进高精度结果输出与迭代统计，保持既有 C 接口兼容。

XianzhiKrylov 26.44

- 公共接口更新为十函数 C API，并保持对 26.43 接口的兼容。
- `create` 只创建计算句柄；JSON 由 `config_set` 加载。
- 增加矩阵、单 RHS 向量和多 RHS 向量对象创建／销毁函数。
- 增加 `xianzhikrylov_execute()`，统一执行分析、数值准备、求解和完整流程。
- 现行 C 接口将求解结果直接写入调用方的解向量对象。
- 矩阵和向量对象使用不透明类型，数据数组所有权仍归调用方。
- Samples 与产品说明文档统一使用十函数调用方式。

XianzhiKrylov 26.43

26.43 的既有接口在 26.45 中继续保持兼容。

XianzhiKrylov 26.42

- 统一开发者预览接口。
- 提供 CSR、BSR、MSR、LDU 和多种 Krylov/预条件器组合。

授权获取 企业微信

store.solved-tech.com