One-sided Jacobi SVD on CUDA GPUs: Profiling-Guided Kernel Fusion and Data-Movement Optimization

郭嘉睿
College of Computer Science, *Nankai University*
guojiarui@mail.nankai.edu.cn

摘要—**Jacobi** 奇异值分解（**Singular Value Decomposition, SVD**）具有良好的数值稳健性和天然的成对并行结构，但朴素 **CUDA** 实现容易把轮次级（**round-level**）控制流直接暴露给主机运行时。本文通过端到端计时、**Nsight Systems** 轨迹和 **Nsight Compute** 内核分析研究一个 **CUDA Jacobi SVD** 实现。初步分析表明，主要瓶颈并非旋转计算本身，而是 **Jacobi** 轮次同步与 **GPU** 执行粒度之间的错配：大量短内核、主机/设备拷贝和同步边界在设备达到高计算利用率或高内存带宽利用率之前已经主导运行时间。随后，本文将评估一种融合协作内核（**fused cooperative kernel**）设计，把轮次级控制推进到生命周期更长的设备端执行上下文中。最终论文将量化加速比、矩阵形状敏感性、剩余瓶颈和可复现性约束。

关键词—**Jacobi SVD**, **CUDA**, **GPU**, 内核融合（**kernel fusion**），协作组（**cooperative groups**），性能剖析（**profiling**）

I. 引言

奇异值分解（Singular Value Decomposition, SVD）是数值分析、科学计算、机器学习、信号处理和低秩近似（low-rank approximation）中的核心稠密线性代数（dense linear algebra）原语 [1]。在这些应用中，SVD 往往不只是一个孤立的数学分解，而是更大计算流水线中的精度、稳定性和数据压缩边界。Jacobi 类 SVD 算法通过反复施加成对正交变换来削弱列之间的相关性；与若干基于双对角化（bidiagonalization）的路径相比，单边 Jacobi SVD（one-sided Jacobi SVD）通常更强调数值稳健性和正交性的保持 [2]。这种性质使 Jacobi SVD 在高精度需求、小中型矩阵批处理以及 GPU 数值库中持续具有现实意义。例如，NVIDIA cuSOLVER 提供基于 Jacobi 方法的 `gesvdj` 接口，并将其定位为适合小中型矩阵的一类 SVD 计算路径 [3]。

然而，算法层面的并行性并不自动转化为 GPU 上的高性能。单边 Jacobi SVD 的一次扫描（sweep）通常由多个轮次（round）组成，每个轮次包含若干互不冲突的列对旋转。列对之间的独立性为并行执行提供了入口，但轮次之间的依赖关系又引入了重复同步点。已有 GPU Jacobi SVD 工作表明，若能让阻塞结构、数据布局和内存层次（memory hierarchy）相匹配，Jacobi SVD 可以有效利用 GPU 资源 [4], [5]。反过来，如果实现把每个轮次直接展开为主机侧 CUDA 运行时调用，那么 GPU 面临的就不再只是浮点运算问题，而是算法执行粒度与设备执行粒度之间的错配。

本文研究一个 CUDA Jacobi SVD baseline。该 baseline 保留了 Jacobi 扫描和轮次结构，但把每个轮次表达为一串由主机控制的设备操作，包括小粒度内核启动、主机/设备数据传输、memset 和同步。这个实现具有很好的诊断价值：它不是一个抽象的伪代码模型，而是许多朴素 GPU 移植会自然走到的形态。也正因为如此，它暴露了一个系统性问题：当每个轮次中的设备端实际工作量不足以摊销 CUDA 运行时固定成本时，大量短内核和同步边界会在 SM 计算吞吐或 DRAM 带宽成为限制因素之前支配端到端运行时间。

本文的中心假设是：该 baseline 的主导瓶颈不是 Jacobi 旋转本身的算术开销，而是轮次级控制面开销（control-plane overhead）。更具体地说，问题不在于 GPU 不能并行处理列对，而在于实现把过细的算法控制流暴露给了主机运行时，使每个 testcase 反复支付内核启动、拷贝、初始化和同步的固定成本。这个假设也决定了优化方向：如果瓶颈来自执行结构，那么只调优单个旋转内核的指令序列、寄存器使用或访存合并（memory coalescing）并不足以解决主要问题；更关键的是提高执行粒度，让更多 Jacobi 轮次在生命周期更长的设备端上下文中完成。

为验证这一假设，本文采用测量驱动的系统研究方法。首先，端到端墙钟时间（wall-clock timing）用于判断不同矩阵形状下的总体行为，并区分高瘦矩阵（tall-skinny matrix）与中等方阵（medium square matrix）对轮次数量的不同压力。其次，Nsight Systems 轨迹用于观察 CUDA API 层面的启动、拷贝、memset 和同步模式；这些累计 API 时间不被简单等同于墙钟时

间，而被用作定位控制面开销的证据。再次，Nsight Compute 采样指标用于检查代表性内核是否已经受限于 SM 吞吐、内存带宽或占用率（occupancy）。最后，本文分析主机侧线程池和流水线并发是否能够掩盖该开销，并区分吞吐量重叠与结构性瓶颈消除这两件不同的事。

在此基础上，本文评估一种融合协作内核（fused cooperative kernel）设计。该设计的目标不是改变 Jacobi SVD 的数学语义，而是改变其执行归属：把轮次级调度和同步从反复返回主机的 CUDA runtime 序列中移出，推进到更粗粒度的设备端执行上下文中。这样做的预期收益有三点。第一，减少每个 testcase 需要支付的内核启动次数。第二，降低轮次间中间状态反复穿越主机/设备边界的机会。第三，使 GPU 在一次较长生命周期的执行中完成更多有用工作，从而缓解短内核主导的低利用率现象。

本文并不主张融合协作内核是 Jacobi SVD 的普遍最优实现。协作内核会受到网格驻留（grid residency）、寄存器压力、共享内存容量、问题规模和设备能力的约束；当矩阵足够大、单轮计算量足够高，或生产级库已经采用更深层的阻塞与批处理策略时，瓶颈可能转向内存层次、数值收敛或负载均衡。本文关注的是一个更窄但常见的工程问题：当朴素 CUDA 映射把 Jacobi 轮次作为主机控制边界时，如何通过性能剖析识别该错误抽象，并用更合适的执行粒度修正它。

本文主要贡献如下：

- 识别一个 CUDA Jacobi SVD baseline 中的轮次级 CUDA 控制面开销，并说明它如何在设备端计算或内存带宽饱和之前成为主导瓶颈。
- 从工作负载形状、列数、Jacobi 轮次数量和 CUDA API 行为四个角度解释该瓶颈为何具有形状敏感性，而不是均匀的常数开销。
- 设计并评估一种融合协作内核执行结构，在保持 Jacobi 扫描语义的同时降低主机运行时交互频率。
- 给出一套可复现的性能剖析流程，结合端到端计时、Nsight Systems 和 Nsight Compute，将控制面开销与设备端吞吐瓶颈区分开来。

图 1 概括了本文研究的核心执行粒度错配。后续章节首先介绍 Jacobi SVD 和 GPU 执行背景，然后给出实验方法、baseline 瓶颈分析、融合协作内核设计、实验评估以及方法边界。

II. 背景与相关工作

A. Jacobi SVD 与 one-sided Jacobi 方法

SVD 是稠密线性代数中的基础分解，经典实现通常在数值稳定性、操作数和目标体系结构之间折中 [1], [6]。Jacobi 类方法的特点是通过一系列成对正交变换逐步降低列相关性。One-sided Jacobi SVD 可追溯到 Hestenes 的双正交化（biorthogonalization）思想 [7]；现代快速 Jacobi SVD 研究进一步说明，pivot 策略、预处理和实现细节会同时影响收敛速度与相对精度 [2]。

本文关心的是 one-sided Jacobi SVD 的执行结构。一次扫描（sweep）由若干轮次（round）组成，每个轮次包含一组不共享列的列对旋转。该结构在列对层面天然并行，但轮次之间存在顺序依赖：下一轮必须看到上一轮更新后的列。若实现把每个轮次都映射为主机侧 CUDA 操作序列，则算法同步点会直接变成运行时同步点。这正是后文 baseline 瓶颈分析要解释的结构来源。

B. GPU 稠密线性代数

GPU 稠密线性代数通常依赖足够大的并行工作量、规则访存和较高算术强度（arithmetic intensity），以摊销 kernel launch、同步和主机/设备交互成本 [8]。因此，一个数学上可并行的算法，若被切成大量短小且相互依赖的阶段，仍可能无法让 GPU 达到高 SM 吞吐或高 DRAM 带宽利用率。

已有 GPU Jacobi SVD 工作强调了算法结构与硬件层次的共同设计。Novaković 的层次阻塞 Jacobi SVD（hierarchically blocked Jacobi SVD）让阻塞结构对应 GPU 内存层次，并研究单 GPU 与多 GPU 上的并行 pivot 策略 [4]。Boukaram 等人的 batched QR/SVD 工作则在小矩阵批处理场景中使用 one-sided Jacobi SVD，因为它简单且具有内在并行性 [5]。这些工作说明，Jacobi SVD 在 GPU 上并非不合适；关键在于执行粒度、数据驻留和同步边界是否匹配硬件。

C. Kernel fusion、cooperative groups 与 CUDA Graphs

当一个计算被拆成许多短 kernel 时，优化方向不一定首先是单个 kernel 内部的指令级调优。更高层的执行结构也可能是主要问题。Kernel fusion 通过把多个逻辑阶段合并到同一个设备端执行上下文中，减少 kernel launch、全局中间状态和主机可见同步点。代价是实现复杂度、寄存器压力、共享内存压力和调度约束可能上升。

CUDA cooperative groups 为更明确的线程协作与同步提供抽象；在满足设备能力和驻留约束时，cooperative kernel 可以支持 grid 范围同步 [8]。这对 Jacobi SVD 有直接意义：传统 CUDA 实现常用 kernel 边界作为 round 间全局同步，而 cooperative kernel 允许部分 round 级同步留在设备端。CUDA Graphs 则提供另一类降低提交开销的机制：把 kernel、memcpy、memset 等操作组织成可重复启动的依赖图，从而降低重复 CPU launch cost [8]。本文选择评估 fused cooperative kernel，是因为我们希望改变

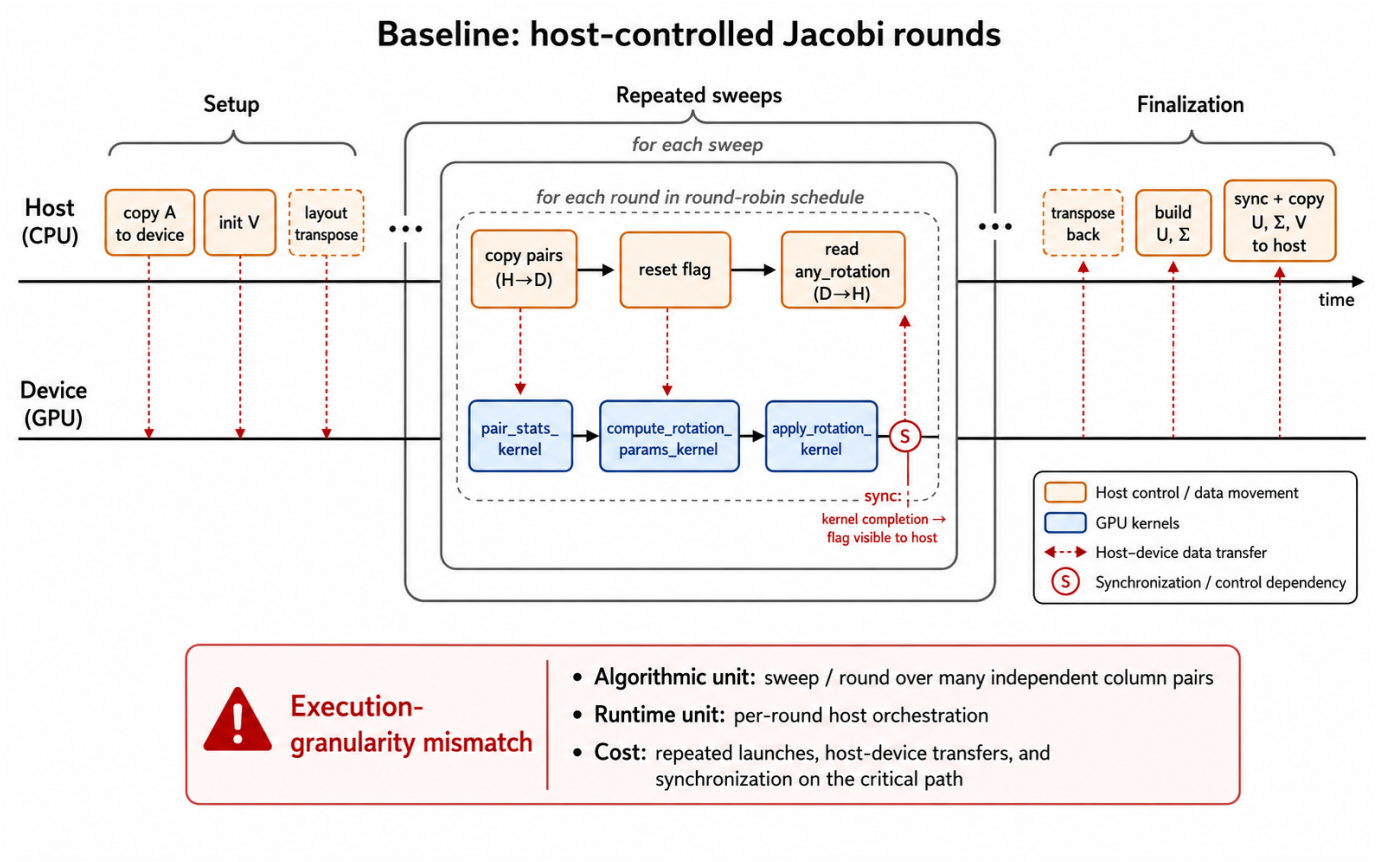

Fig. 1. Baseline 实现中的执行粒度错配：Jacobi round 被展开为主机侧拷贝、初始化、短内核启动和同步边界，导致控制面开销在设备端计算饱和前主导端到端时间。

round 级控制所在的位置，而不仅是压缩一串既有主机提交。

D. Profiling-guided GPU optimization

本文采用性能剖析驱动优化（profiling-guided optimization），因为同样的端到端慢速可能来自不同原因：扫描次数过多、单个旋转 kernel 低效、访存带宽受限、kernel 过短、memcpy/memset 过多，或同步边界过细。仅看墙钟时间无法区分这些情况。

Nsight Systems 和 Nsight Compute 在本文中承担互补角色。Nsight Systems 用于观察 CUDA API 行为，包括 kernel launch、memcpy、memset 和同步模式 [9]。Nsight Compute 用于采样代表性 kernel 的微体系结构（microarchitecture）指标，例如占用率（occupancy）、SM 吞吐和内存吞吐 [10]。后文将结合两类证据判断 baseline 是否真的受限于设备端计算/带宽，还是主要受限于 round-level CUDA control-plane overhead。

III. 实验方法

本章定义本文所有性能与正确性结论所依赖的实验协议。为了避免把测量噪声误读为算法性质，实验方法被拆成七个互不重叠的部分：硬件与软件环境、工作负载构造、端到端计时、性能剖析、主机并发与消融、正确性检查，以及可复现性限制。正文只描述实验设计、测量边界和归因原则；可执行文件、脚本、命令行开关、归档目录等工程标识符由实现细节与可复现性附录中的 manifest 约定记录，见附录 A 和附录 C。除非另有说明，baseline 与优化实现使用相同输入矩阵、相同收敛容差、相同最大扫描次数和相同编译配置；两者之间唯一被刻意改变的因素是 Jacobi 轮次控制和数据移动的执行结构。

A. 硬件与软件环境

所有实验均在单 GPU 节点上执行。归档环境报告和终端复查显示，主机侧环境为 Ubuntu 24.04.4 LTS on WSL2，Linux kernel 为 6.6.87.2-microsoft-standard-WSL2，CPU 为 12th Gen Intel Core i9-12900H，共 20 个逻辑 CPU。被测 GPU 为 NVIDIA GeForce RTX 3070 Ti Laptop GPU，compute capability 为 8.6，包含 46 个 SM，显存为 8192 MiB，并支持协作启动（cooperative launch）。NVIDIA driver 版本为 572.61，`nvidia-smi` 报告的 CUDA runtime 版本为 12.8；CUDA Toolkit 版本为 12.8，`nvcc` 为 V12.8.93。

构建与剖析软件栈包括 GCC/G++ 13.3.0、CMake 3.28.3、Ninja 1.11.1、GNU Make 4.3、Nsight Systems 2024.6.2 和 Nsight Compute 2025.1.1。本文把 WSL2 也列为实验环境的一部分，而不是把它视为透明层：GPU 计算仍由 NVIDIA 驱动暴露给 Linux 用户态，但文件系统、进程调度和 host-side timing 可能受虚拟化边界影响。因此，后续结论只声称在该软硬件栈下成立，跨裸机 Linux、不同驱动或不同 GPU 架构的外推需要重新测量。

计时运行与性能剖析运行由同一套 benchmark harness 生成，并分别保存为 baseline 与融合实现两组归档实例；归档实例通过 manifest 暴露运行配置和产物索引，组织原则见附录 A 和附录 C。若某个 GPU 不满足协作启动或网格驻留（grid residency）要求，优化实现必须显式回退到 baseline 或标记该配置不适用，而不能静默改变问题规模。

B. 工作负载

基准集合覆盖 14 个计时工作负载，用于分别施压 Jacobi SVD 的列对并行性、轮次数量和主机运行时调用频率。表 I 列出实际工作负载集合。baseline 与融合实现复用同一批 MAT/TXT 输入流；输入集合标识和解析规则由归档 manifest 记录，格式约定见附录 A。工作负载包括小型网格（small grid）、中等方阵（medium square matrix）、高瘦矩阵（tall-skinny matrix）、病态矩阵（ill-conditioned matrix）、低秩矩阵（low-rank matrix）、稀疏矩阵（sparse matrix）和零列矩阵（zero-column matrix）。这样的划分不是单纯的数据集枚举，而是为了回答一个结构性问题：端到端耗时是否随 Jacobi 轮次结构变化，而不是只随矩阵元素总数变化。

每个工作负载由输入格式、矩阵维度、testcase 数量、布局转置（layout transpose）策略、收敛容差和最大扫描次数共同定义。本次计时实例统一限制为 32 个最大 sweep、每个 block 256 个线程和容量为 2 的输出队列；这些取值作为运行配置字段由归档 manifest 记录。三种布局策略只对中等方阵与中等高瘦矩阵成组展开，用于分离列连续访问对端到端时间的影响。MAT 输入被视为二进制 testcase stream，TXT 输入被视为文本矩阵集合；正文只报告由归档输入解析得到的 testcase 数量。

C. 计时测量

主要性能指标是端到端墙钟时间（end-to-end wall-clock time）。计时区间从工作负载进入被测 SVD 路径之前开始，到最后一个 testcase 完成并经过必要的设备同步之后结束。因此，该指标包含 kernel launch、主机/设备拷贝、设备端计算、内存初始化、同步、主机侧调度和线程池开销。本文选择端到端计时作为主指标，是因为研究对象正是 CUDA runtime 控制面开销；若只测量单个 kernel 的设备端耗时，会系统性低估 baseline 的真实成本。

端到端计时对 baseline 与融合实现分别使用一组已归档实例。每个实例包含表 I 中的 14 个工作负载，每个工作负载重复 3 次，因此每个实现有 42 条计时记录；全部记录均成功完成。单个工作负载的报告值采用中位数（median）作为主要统计量，并辅以均值（mean）和标准差（standard deviation）描述运行间波动。baseline 与优化实现的加速比定义为

$$S_w = \frac{T_{\mathrm{baseline},w}}{T_{\mathrm{optimized},w}}, \tag{1}$$

其中 w 表示工作负载，T 表示该工作负载的中位端到端时间。跨工作负载汇总使用几何平均（geometric mean）：

$$S_{\mathrm{geo}} = \exp\left(\frac{1}{|W|}\sum_{w\in W}\log S_w\right). \tag{2}$$

几何平均不会让绝对耗时很长的单个 workload 独占结论，因此更适合汇总一组比例型 speedup。

若使用 CUDA event 进行辅助测量，本文只把它作为设备端阶段的局部诊断，而不作为端到端主结果。CUDA event 可测量两个 event 之间的 GPU 时间 [8]，但它不能自然覆盖主机侧排队、CPU 线程调度、运行时 API 调用重叠和跨 testcase 编排（orchestration）；这些恰好是本文要分析的成本。

D. Nsight Systems 性能剖析

Nsight Systems 用于捕获系统级时间线（system-level timeline）和 CUDA API trace，包括 kernel 启动、协作 kernel 启动、主机/设备拷贝、设备内存初始化、设备同步以及 stream 同步事件 [9]。本文从 trace 中提取三类信息：第一，CUDA API 调用次数；第二，各类 API 的累计 host 端耗时；第三，GPU kernel 与数据传输在时间线上的排列关系。对于 baseline，重点观察每个 Jacobi round 是否对应重复的“初始化、启动、拷贝、同步”模式；对于优化实现，重点观察这些操作是否被提升到 sweep 级或 testcase 级边界。

CUDA API 累计时间只被解释为开销归因信号，而不是墙钟时间替代物。原因是多线程主机提交、异步 CUDA stream 和 profiler 采样窗口会让多个 API 活动在时间上重叠；累计 API 时间可能大于进程的实际 wall-clock time。本文因此把 Nsight Systems 证据用于回答“开销堆在哪里”，而不是直接回答“程序跑了多久”。实际加速结论仍以第 3.3 节的端到端计时为准。

为降低性能剖析器（profiler）扰动，Nsight Systems 运行与正式计时运行分开执行。性能剖析实例对 10 个代表性工作负载执行一次系统级 trace，并配套

TABLE I
实验使用的基准工作负载。维度与 TESTCASE 数量来自归档 BENCHMARK 配置及其输入文件。

工作负载	输入	形状类别	Layout	行数	列数	Testcases
mat_grid_small_auto	MAT	小型网格（small grid）	auto	8–80	4–64	96
mat_square_medium_auto	MAT	中等方阵（medium square）	auto	32–256	32–256	48
mat_square_medium_off	MAT	中等方阵（medium square）	off	32–256	32–256	48
mat_square_medium_on	MAT	中等方阵（medium square）	on	32–256	32–256	48
mat_tall_skinny_medium_auto	MAT	高瘦矩阵（tall-skinny）	auto	128–1024	8–64	64
mat_tall_skinny_medium_off	MAT	高瘦矩阵（tall-skinny）	off	128–1024	8–64	64
mat_tall_skinny_medium_on	MAT	高瘦矩阵（tall-skinny）	on	128–1024	8–64	64
mat_ill_conditioned_auto	MAT	病态矩阵（ill-conditioned）	auto	128–1024	8–64	40
mat_low_rank_auto	MAT	低秩矩阵（low-rank）	auto	64–512	8–64	40
mat_sparse_auto	MAT	稀疏矩阵（sparse）	auto	64–512	8–64	40
mat_zero_columns_auto	MAT	零列矩阵（zero-column）	auto	64–512	8–64	40
txt_grid_small_auto	TXT	小型网格（small grid）	auto	2768	4	1
txt_ill_conditioned_small_auto	TXT	病态矩阵（ill-conditioned）	auto	3904	8	1
txt_sparse_small_auto	TXT	稀疏矩阵（sparse）	auto	4768	16	1

两种 Nsight Compute 采样深度。profile 集合覆盖 grid、square、tall-skinny、ill-conditioned、sparse 与 TXT grid，但不包括 low-rank、zero-columns、TXT ill-conditioned 和 TXT sparse；因此 profiling 证据用于解释控制面结构，不用于替代表 I 的完整端到端计时集合。

E. Nsight Compute 性能剖析

Nsight Compute 用于采样代表性 kernel 的微体系结构（microarchitecture）行为 [10]。本文关注的指标包括 kernel duration、achieved occupancy、SM throughput、DRAM throughput、memory dependency、warp stall reason 和 launch 配置。采样对象覆盖 baseline 中最频繁出现的 Jacobi 旋转阶段，以及优化实现中的融合协作 sweep。若某个工作负载产生大量同类短 kernel，Nsight Compute 不对所有 kernel 做全量 profiling，而是通过固定过滤规则选择代表性样本；过滤规则作为 profiler 配置字段由归档 manifest 记录。

Nsight Compute 的作用是排除错误解释。若采样 kernel 的 SM throughput 和 DRAM throughput 均较低，且单个 kernel duration 很短，则端到端慢速更可能来自过细执行粒度、launch overhead 或同步边界，而不是设备端算术吞吐已被打满。反过来，若 fused kernel 使单次执行时间显著变长并提高资源利用率，说明优化至少部分成功地把工作从主机控制面推回了设备执行面。由于 Nsight Compute 会重放 kernel 并改变运行时行为，本文不把其结果作为端到端时间来源。

F. 主机并发与消融测量

本文单独测量主机侧线程池和流水线并发（pipeline concurrency）的影响。该测量回答的问题是：并发提交是否只是重叠多个独立 testcase 的等待时间，还是改变了单个 testcase 内部的 Jacobi round 级结构。为此，实验比较不同在途任务容量、不同 worker 数量和单线程执行模式下的端到端时间，并结合 Nsight Systems 中 CUDA Runtime API 活动跨度与累计 API 时间估计 host-side overlap。

消融实验（ablation study）按执行结构因素分组，而不是按代码补丁历史分组。候选消融项包括：主机并发、设备缓冲区复用、主机/设备传输减少、round 内部融合，以及完整 fused cooperative kernel。若某一项无法在实现中干净隔离，本文将明确说明该消融是组合效应，避免把多个变化归因到单一优化。

G. 正确性检查

正确性检查从三个层面进行。第一，比较 baseline 与优化实现的收敛行为，包括每个 testcase 的扫描次数、是否达到相同容差、是否触发最大扫描次数上限。第二，计算重构误差（reconstruction error）：

$$\epsilon_{\mathrm{rec}} = \frac{\|A - U\Sigma V^T\|_F}{\|A\|_F}, \tag{3}$$

其中 $\|\cdot\|_F$ 为 Frobenius 范数（Frobenius norm）。第三，计算正交性误差（orthogonality error）：

$$\epsilon_U = \frac{\|U^TU - I\|_F}{\|I\|_F}, \qquad \epsilon_V = \frac{\|V^TV - I\|_F}{\|I\|_F}. \tag{4}$$

若实现只显式生成 V 或只返回奇异值，则相应地报告可观测部分，并说明缺失量无法直接验证。

优化实现被认为保持数值行为，需要满足两个条件：其扫描次数与 baseline 一致，或差异可以由浮点舍入和并行规约顺序解释；其重构误差与正交性误差不劣于同精度下可接受的数值范围。由于 fused cooperative kernel 可能改变中间同步位置和浮点操

作顺序，本文不要求逐 bit 相同，而要求数学误差和收敛判定保持一致。

H. 可复现性与有效性威胁

附录 C 定义计时命令、Nsight Systems 命令、Nsight Compute 命令、环境变量、随机种子、原始输出文件和统计脚本位置等信息应如何进入归档 manifest。正文只报告足以支撑结论的汇总表和图，完整数据放入可复现性归档，以便复查异常值。

本文实验的有效性威胁（threats to validity）主要来自五个方面。第一，单 GPU 平台不能代表所有架构，尤其是 cooperative kernel 驻留约束可能随 GPU 代际变化。第二，profiler 会引入扰动，因此性能剖析运行（profiling run）与计时运行（timing run）必须分离。第三，动态频率、温度和后台进程可能影响 wall-clock time。第四，工作负载集合若只覆盖少数矩阵形状，就不能推出对所有 Jacobi SVD 场景的普遍结论。第五，CUDA API 累计时间在多线程提交下不能直接等同于端到端时间。本文把这些限制显式纳入方法章节，是为了让后续性能结论保持可证伪、可复查。

IV. Baseline 瓶颈分析

本章的目标不是简单判定 baseline 性能不足，而是分解其主要成本来源：它究竟来自设备端算术、设备端内存带宽、主机/设备数据移动，还是 CUDA runtime 的控制面开销(control-plane overhead)。这里采用一个互斥且尽量穷尽的归因顺序：先看端到端计时是否随矩阵形状系统性变化，再用 Nsight Systems 定位 API 调用堆积，随后用 Nsight Compute 排除 SM 和 DRAM 饱和，最后讨论线程池并发为何只能缓解吞吐量而不能消除结构性瓶颈。

A. 形状相关的计时行为

第一个诊断问题是：减速是否在所有工作负载上均匀出现，还是与矩阵形状相关。baseline timing 归档包含 14 个工作负载、每个工作负载 3 次运行，共 42 条成功记录。若主要瓶颈只是输入解析、固定启动成本或随机测量噪声，那么不同形状之间的按 testcase 归一化时间不应出现稳定数量级差异；实际数据则相反。

中等方阵组 `mat_square_medium_*` 是最重的一组。其应用报告平均耗时为 6913.9–7230.8 ms，按 testcase 归一化后为 144.04–150.64 ms/testcase。相比之下，中等高瘦矩阵组 `mat_tall_skinny_medium_*` 为 1396.4–1446.6 ms,按 testcase 归一化后为 21.82–22.60 ms/testcase。也就是说，即使消除 testcase 数量差异，方阵组仍慢约 6.4×–6.9×。

这个差异不能只用“收敛更困难”解释。`mat_ill_conditioned_auto` 和 `mat_low_rank_auto` 的平均 sweep/testcase 分别为 11.30 和 10.07，并不低；但它们的平均耗时只有 33.56 和 29.34 ms/testcase，远低于中等方阵组。更好的解释是矩阵列数改变了 Jacobi 轮次结构：单边 Jacobi SVD（one-sided Jacobi SVD）在每个 sweep 内按轮转调度（round-robin scheduling）处理列对；列数越大，round 数量和每个 round 周围的固定 runtime 操作越多。这里暴露的不是传统意义上的单个大 kernel 算得慢，而是算法同步粒度与 GPU 执行粒度之间的错配。

B. CUDA API 数量膨胀

Nsight Systems 进一步说明，方阵组变慢不是因为少量长 kernel 占据了设备，而是因为 baseline 为每个 Jacobi round 支付了细粒度主机控制成本。表 II 汇总了 10 个 profile 工作负载的 CUDA API 行为。这里的 API 时间是 Nsight Systems 对 host-side API duration 的求和，不等同于 wall-clock time；由于线程池可能让多个主机线程重叠提交 CUDA work，它只能作为“开销堆在哪里”的证据，而不能直接当成程序运行时间。

`mat_square_medium_auto` 的规模最能说明问题：一次 profile 中出现 232,692 次内核启动（kernel launch）、155,200 次 `cudaMemcpy` 和 77,504 次 `cudaMemset`。这些数字近似满足

$$N_{\text{launch}} \approx 3N_{\text{round}},$$
$$N_{\text{memcpy}} \approx 2N_{\text{round}},$$
$$N_{\text{memset}} \approx N_{\text{round}},$$

其中 N_{round} 可由 `cudaMemset` 次数近似估计。这与归档 baseline 的 round 级执行模式吻合：每个 round 需要把当前列对从主机拷到设备（host-to-device copy, H2D），清零一个设备端收敛标志，启动若干短 kernel 完成统计和旋转更新，再把是否发生旋转的标志从设备拷回主机（device-to-host copy, D2H）。换句话说，baseline 把 Jacobi 的 round 级控制流直接暴露给 CUDA runtime，导致大量微小 GPU work 被 host-side launch、copy、memset 和同步边界包围。

这个比例关系也解释了形状敏感性。高瘦矩阵组的 `cudaMemset` 次数约为 14,684，而中等方阵组为 77,504，后者约为前者 5.3×。两组的按 testcase 耗时差异约 6.4×–6.9×，数量级上与 round-like 单元数的放大一致。因此，当前最有解释力的变量不是元素总数本身，而是列数诱发的 round 数量和 round 边界固定成本。

C. 排除设备端饱和

如果 baseline 的主要问题是设备端计算吞吐不足，那么 Nsight Compute 应该看到长时间运行的核心 kernel、较高的流式多处理器（Streaming Multiprocessor, SM）吞吐，或者至少某些 kernel 接

TABLE II
BASELINE CUDA API 行为的 NSIGHT SYSTEMS 汇总。单元格格式为累计 API 时间秒 / 调用次数。

工作负载	cudaLaunchKernel	cudaMemcpy	cudaMemset	cudaMalloc+Free
mat_grid_small_auto	2.755 / 56,502	2.300 / 37,920	1.102 / 18,768	0.168 / 2,118
mat_ill_conditioned_auto	1.613 / 43,444	1.538 / 29,032	0.594 / 14,436	0.105 / 936
mat_sparse_auto	0.883 / 24,207	0.890 / 16,214	0.352 / 8,027	0.113 / 926
mat_square_medium_auto	13.332 / 232,692	10.630 / 155,200	5.548 / 77,504	0.141 / 1,140
mat_square_medium_off	13.317 / 232,608	11.202 / 155,200	5.818 / 77,504	0.159 / 1,056
mat_square_medium_on	12.488 / 232,704	10.431 / 155,200	5.059 / 77,504	0.155 / 1,152
mat_tall_skinny_medium_auto	1.369 / 44,268	1.468 / 29,624	0.505 / 14,684	0.132 / 1,496
mat_tall_skinny_medium_off	1.423 / 44,180	1.557 / 29,624	0.515 / 14,684	0.110 / 1,408
mat_tall_skinny_medium_on	1.449 / 44,308	1.548 / 29,624	0.497 / 14,684	0.155 / 1,536
txt_grid_small_auto	1.569 / 29,216	1.304 / 19,648	0.606 / 9,696	0.525 / 1,408

近设备能力上限。如果主要问题是全局内存带宽，动态随机存取内存吞吐（DRAM throughput）也应明显升高。采样结果并不支持这两种解释。

baseline profile 的 Nsight Compute 样本共覆盖 30 个 kernel 记录。被采样的 `pair_stats_kernel` 持续时间通常约 6.4–7.0 μs，`apply_rotation_kernel` 约 4.3–5.2 μs；所有样本的持续时间范围为 4.352–7.008 μs，中位数为 6.496 μs。SM throughput 的范围为 0.24%–10.77%，中位数仅 1.985%；DRAM throughput 的范围为 0.19%–7.39%，中位数为 0.925%。

这些数值说明 GPU 端工作粒度过细：kernel 持续时间很短，硬件吞吐尚未被充分填满，控制权就回到主机侧。于是 Nsight Compute 与 Nsight Systems 的证据互相扣合：不是设备端算术或 DRAM 带宽已经饱和，而是 baseline 没有给 GPU 足够粗的连续执行单元。

D. 线程池并发

线程池并发（thread-pool concurrency）提供了一个有用的反例。流水线可以把多个 testcase 的等待时间重叠起来，因此它可能降低一批输入的 wall-clock time；但它不改变单个 testcase 内部的 round 级结构。换言之，并发可以提高吞吐量，却不能让每个 round 的 H2D、memset、kernel launch 和 D2H 往返消失。

从系统角度看，这一点很关键。若把线程池看成“优化”，它更像是吞吐放大器，而不是根因修复。对于当前 baseline，大量短 kernel 和小拷贝已经给 CUDA runtime/driver 制造了高频提交压力；多个 CPU worker 同时向同一 GPU 提交这种细碎 work，既可能隐藏等待，也可能进一步增加调度拥塞。因此，线程池实验应被解释为 pipeline 层的重叠能力，而不是 Jacobi SVD 设备端执行结构已经合理的证据。

E. 瓶颈小结

本章得到的瓶颈归因可以按 MECE 原则收束为四类。第一，计算瓶颈证据弱：采样 kernel 只有数微秒，SM throughput 中位数约 1.985%。第二，设备端 DRAM 带宽瓶颈证据弱：DRAM throughput 中位数约 0.925%，远未表现为带宽墙。第三，内存分配是次级成本：各工作负载中 `cudaMalloc`+`cudaFree` 通常只有数百到一千多次，累计 API 时间明显低于 launch、copy 和 memset，但后续 workspace 复用仍可能有收益。第四，CUDA 调度与同步瓶颈证据最强：中等方阵组出现约 77.5k 个 round-like 单元，并放大为约 232k 次 kernel launch、155k 次 memcpy 和 77.5k 次 memset。

因此，baseline 的核心问题不是某个 kernel 的局部算术效率不足，而是执行结构把 Jacobi round 这种算法级同步边界机械地投射到了 CUDA runtime 边界上。后续优化应优先改变控制粒度：把 round 级主机控制提升为 sweep 级或更粗粒度的设备端执行，而不是先在微秒级 kernel 内做局部算术调优。这个结论也为第 5 章的 fused cooperative kernel 设计提供了直接动机。

V. 融合协作内核设计

第 4 章的瓶颈归因表明，baseline 的主要损失不来自单个旋转公式，而来自执行边界选择：Jacobi round 被逐一暴露为主机侧 CUDA runtime 操作，导致数学上的细粒度同步点被放大为 kernel launch、memcpy、memset 和设备到主机回读（device-to-host copy, D2H）。融合协作内核（fused cooperative kernel）的设计目标因此不是改变 one-sided Jacobi SVD 的数值算法，而是改变控制权所在的位置：保留 sweep 级收敛判定在主机可见边界上，同时把 round 级列对生成、统计量计算、旋转应用和 round 间同步放入一个设备端执行上下文。这样做对应一种更粗粒度的

不变量：主机观察到的是“一个 sweep 是否发生过旋转”，而不是“每一个 round 内部如何提交”。

A. 设计目标与边界

设输入矩阵为 $A \in \mathbb{R}^{m\times n}$，且当前实现要求 $m \geq n$。One-sided Jacobi SVD 通过右乘一系列平面旋转（plane rotation）逐步正交化 A 的列；同一旋转也累积到 $V \in \mathbb{R}^{n\times n}$。对任意列对 (p, q)，一次更新只依赖三项局部统计量

$$a_{pp} = A_{:,p}^T A_{:,p}, \qquad a_{qq} = A_{:,q}^T A_{:,q}, \\ a_{pq} = A_{:,p}^T A_{:,q}. \tag{5}$$

若

$$|a_{pq}| > \epsilon\sqrt{\max(a_{pp}, 0)\max(a_{qq}, 0)} \tag{6}$$

且 $|a_{pq}|$ 不落入实现设置的极小除数保护区，则计算 Jacobi 旋转参数 (c, s) 并同时更新 A 与 V 的第 p, q 列。该判定与旋转公式沿用 baseline；融合路径只改变这些操作的编排位置。

该设计有三个约束。第一，不能破坏 Jacobi sweep 的顺序语义：第 $r+1$ 个 round 必须看到第 r 个 round 完成后的列值。第二，不能在同一 round 中引入列写冲突；否则设备端并行性会改变算法定义。第三，不能把 cooperative launch 的硬件前提伪装成普遍可用能力；当设备不支持协作启动或 grid 无法满足驻留约束时，实现必须回退到 baseline 路径。换言之，本章讨论的是一个受控替换：改变控制粒度，不改变可观察的数值问题。

B. 从 *Round* 级流水到 *Sweep* 级设备上下文

Baseline 的一个 sweep 可以抽象为三层循环：外层枚举 sweep，中层枚举 round，内层并行处理该 round 的无冲突列对。baseline 的工程问题在于，中层循环由主机驱动：每个 round 都需要主机准备列对数组、清零设备端收敛标志、启动统计 kernel、启动旋转 kernel，并回读该 round 是否发生旋转。于是，数学上必要的 round 间同步被实现为主机可见的 runtime 边界。

融合路径把中层循环下沉到 cooperative kernel 内部。主机在每个 sweep 只执行一个短序列：清零 sweep 级标志、发起一次 cooperative kernel、回读该 sweep 的标志。设备端 kernel 内部执行所有 round；每个 round 结束后使用协作组（cooperative groups）提供的全网格同步（grid-wide synchronization）作为 round 间屏障 [8]。图 2 给出了这一边界重排。

这种重排并不消除同步本身。Jacobi 方法的顺序依赖仍然存在，且每个 round 后仍需全局屏障。真正被移除的是主机参与该屏障的方式：round 边界从 CUDA runtime 边界变为设备端 grid 同步边界。由于第 4 章已经排除了 SM 与 DRAM 饱和作为主因，这种边界迁移比局部指令优化更贴近实测瓶颈。

C. 无冲突列对调度

融合内核使用与 baseline 等价的巡回赛调度（round-robin schedule）。令

$$n' = n + (n \bmod 2), \qquad R = n' - 1, \qquad P = n'/2, \tag{7}$$

其中 n' 为加入虚拟列（dummy column）后的偶数列数，R 是每个 sweep 的 round 数，P 是每个 round 的列对槽数量。若 n 为奇数，虚拟列只参与调度占位，不产生真实旋转。每个 round 固定一个首元素，并旋转其余 $n' - 1$ 个位置；第 r 个 round 的第 i 个槽与对称位置配对。实现中设备端按位置重新计算列索引，而不是从主机复制整张调度表。

该调度的核心性质是：

$$\forall r, \quad (p_i, q_i), (p_j, q_j) \in \mathcal{P}_r,\ i \neq j \\ \Rightarrow \{p_i, q_i\} \cap \{p_j, q_j\} = \emptyset. \tag{8}$$

其中 $\mathcal{P}_r$ 表示第 r 个 round 的真实列对集合。式 (8) 使 round 内列对可以并行执行：不同 block 更新的 A 列和 V 列互不重叠，因此不存在写写冲突（write-write conflict）。同时，round 之间仍按 $r = 0, \ldots, R-1$ 顺序执行，并由 grid 同步保证可见性。这一点是融合设计的正确性支点：我们利用的是 round 内交换性，而不是假设所有 Jacobi 旋转都可任意重排。

D. *Block* 到列对的映射

融合 kernel 采用“一个 block 一次处理一个列对槽”的基本映射。给定 grid 大小 G，block b 在一个 round 内处理槽

$$b,\ b + G,\ b + 2G, \ldots < P. \tag{9}$$

因此当可驻留 block 数小于列对槽数时，内核不会扩大 grid 到不可协作同步的规模，而是让每个 block 在同一 round 内串行处理多个无冲突槽。这个选择牺牲一部分 round 内并行度，换取 cooperative kernel 的全网格同步合法性。

每个列对的设备端执行由三个阶段组成。首先，block 内线程沿行维度分片读取 $A_{:,p}$ 与 $A_{:,q}$，并通过共享内存归约（shared-memory reduction）得到 a_{pp}、a_{qq} 和 a_{pq}。其次，单个线程根据式 (6) 计算旋转参数，并把 (c, s) 广播给同一 block。最后，block 内线程再次沿行维度更新 A 的两列，并沿 V 的行维度更新 V 的两列。若判定不需要旋转，则该列对只完成统计和同步，不写回列数据。

这种映射保持了两个局部性原则。第一，列对是调度的原子单位；旋转参数只在处理该列对的 block 内共享，不需要全局临时数组保存所有 a_{pp}, a_{qq}, a_{pq}。第二，round 的完成条件是所有 block 完成其负责的槽集合；因此一个 grid 同步就足以替代 baseline 中围绕每个 round 的主机侧 launch 和回读序列。

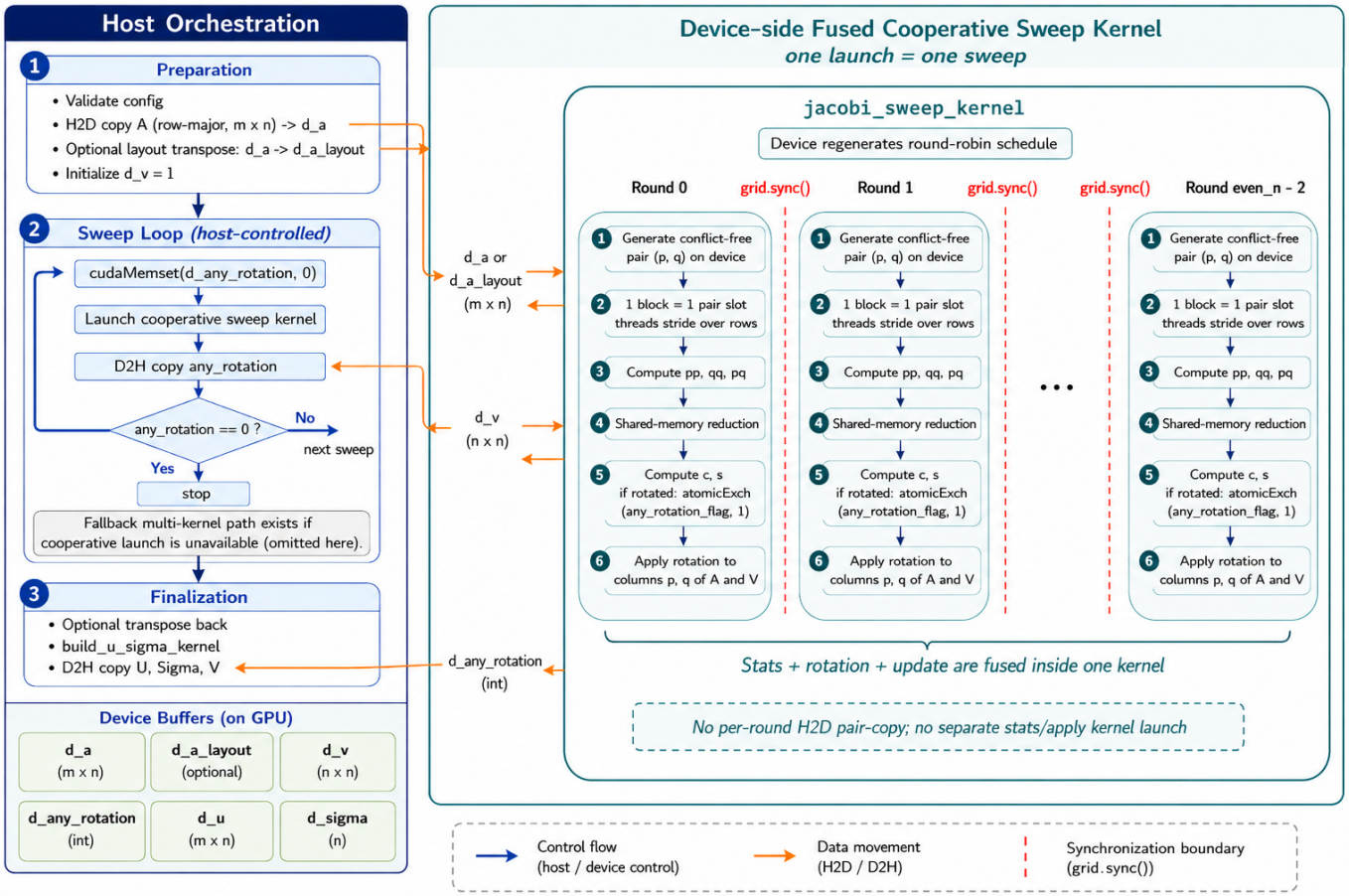

Fig. 2. 融合协作内核的执行边界重排：主机保留 sweep 级控制与收敛回读，设备端在单个 cooperative kernel 中完成 round 调度、列对处理和 round 间全网格同步。

E. 驻留约束与 *Grid* 规模

Cooperative kernel 的全网格同步要求参与同一 grid 的所有 block 能够同时驻留（reside）在 GPU 上 [8]。因此，grid 大小不是简单取 P，而是由资源约束裁剪。实现先查询设备是否支持 cooperative launch，再通过 occupancy 查询得到每个 SM 可同时驻留的 block 数。若 GPU 有 S 个 SM，每个 SM 可驻留 B_{SM} 个该 kernel 的 block，则可协作同步的 block 上限为

$$B_{\mathrm{res}} = SB_{\mathrm{SM}}. \tag{10}$$

实际 grid 大小取

$$G = \min(B_{\mathrm{res}}, P). \tag{11}$$

式 (11) 把硬件驻留约束变成算法调度的一部分：若列对槽数量超过可驻留 block 数，剩余槽由已驻留 block 以 stride 方式处理；若驻留上限为零，或设备不支持 cooperative launch，则该路径不可用。

共享内存使用也进入同一约束。每个 block 至少需要 $3T$ 个 double 的动态共享内存来保存三项归约数组，其中 T 是每个 block 的线程数；另有少量共享标量用于广播旋转参数。线程数先被限制到 CUDA 合法范围并按 warp 对齐，再参与 occupancy 查询。这样设计的目的不是追求最大理论占用率（occupancy），而是确保同步语义和资源语义一致：只有能够整体驻留的 grid 才允许执行 grid 范围屏障。

F. 数据布局与访存路径

主机输入和最终输出保持行主序（row-major），这是接口层的稳定约定。Jacobi 旋转却反复访问矩阵列；当 A 保持行主序时，沿行扫描同一列会形成跨步访问（strided access）。优化实现因此提供可选的布局转置（layout transpose）路径：在进入 Jacobi sweep 前把 A 转为列连续布局，在 sweep 结束后再转回行主序以构造 U 和输出结果。该路径不改变 V 的布局，V 仍按行主序更新。

从设计角度看，布局转置是独立于 cooperative fusion 的第二个优化维度。融合处理的是控制粒度和同步边界；转置处理的是列访问的内存连续性。两者通过统一的索引策略相连：计算主体以逻辑坐

标 (row, col) 描述访问，而物理地址由布局策略决定。这样避免在核心旋转逻辑中散落布局分支，也使 auto/off/on 三种布局实验能解释为同一算法在不同物理布局下的对比。

G. 收敛标志与数值等价

融合路径把 baseline 的 round 级收敛标志提升为 sweep 级标志。任意列对发生有效旋转时，设备端通过原子写把该 sweep 的标志置为 1；整个 cooperative kernel 结束后，主机只回读一次该标志。若标志仍为 0，说明该 sweep 的所有真实列对均未通过式 (6)，执行可以提前停止。这个改变减少了 D2H 回读次数，但不改变停止条件的逻辑含义：baseline 在一个 sweep 内只要观察到任一 round 发生旋转，也会继续下一 sweep。

数值等价建立在三个层面。第一，列对集合与 baseline 的 round-robin schedule 等价，只是由设备端按公式重算。第二，每个列对使用相同的 pivot 判定和同类稳定旋转公式；当前实现还保留 $|a_{pq}| > 10^{-300}$ 的保护，以避免极小除数造成不稳定参数。第三，同一 round 内的列对作用于互不相交的列，因而在实数算术下可交换；在浮点算术中，它们也不会竞争同一存储位置。因此融合路径不承诺逐 bit 相同，而承诺保持 sweep 语义、收敛判定和误差指标的可比性。第 6 章的重构误差和正交性误差用于检验这一承诺。

H. 回退路径与有效性边界

融合协作内核是优化路径，不是唯一正确路径。以下条件会触发回退：$n < 2$，设备不支持 cooperative launch，occupancy 查询无法给出正的驻留 block 数，或当前线程数与共享内存配置无法形成合法 cooperative grid。回退后执行 round 级 baseline：主机生成列对调度，并按 round 提交统计和旋转 kernel。该回退保持可移植性，也让实验能够区分“优化不可用”和“算法失败”。

这种设计仍有明确边界。它只把 round 级控制移到设备端，并未消除 sweep 级主机循环；原因是 sweep 级提前停止需要一个全局收敛判定，而当前实现选择保留主机可见的 testcase 级控制结构。它也没有解决所有固定成本：设备缓冲区分配、输入输出拷贝、布局转置和最终 U, Σ, V 构造仍在端到端计时中。正因为这些成本没有被隐藏，本章的融合设计应被理解为对第 4 章最强瓶颈的定向修复，而不是 GPU Jacobi SVD 的理论下限。

VI. 实验评估

实验评估回答四个问题：

1) 融合设计获得多少端到端加速？
2) 哪些工作负载收益最大，原因是什么？
3) 优化是否保持数值行为？

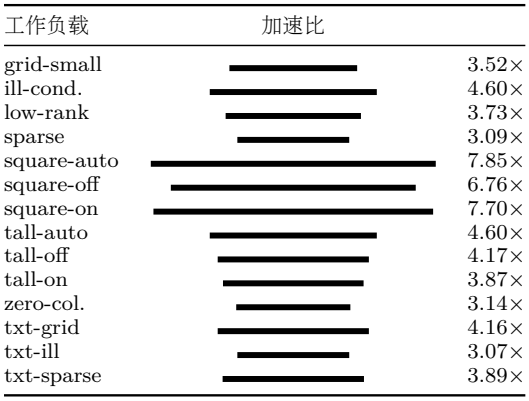

Fig. 3. 各工作负载的端到端应用耗时加速比。条形长度按最大观测加速比 7.85× 归一化。

4) 融合之后还剩下哪些瓶颈？

本章按预备骨架组织，只陈述实验观测和直接归因，不展开设计取舍、适用边界和后续优化策略；这些解释性内容放在第 7 章讨论中。所有端到端时间来自 2026-05-15 归档的 baseline 与 fused timing 实例；性能剖析来自同日归档的 Nsight Systems 和 Nsight Compute 结果。除非另有说明，时间使用应用报告（application report）中的 `elapsed_ms`，而不是 profiler 的 CUDA API 累计时间。

A. 端到端加速

表 III 给出 14 个工作负载的端到端结果。融合实现相对 baseline 在所有工作负载上均取得加速，应用耗时几何平均加速为 4.36×；若改用外部 wall-clock median 计算，几何平均加速为 4.31×，与应用报告口径一致。加速比范围为 3.07×–7.85×，说明优化不是只改善单个异常 case，而是在完整 benchmark 集合上都改变了执行成本结构。

最重的 `mat_square_medium_*` 组收益最大：`auto`、`off` 和 `on` 分别达到 7.85×、6.76× 和 7.70×。按 testcase 归一化后，`mat_square_medium_auto` 从 148.09 ms/testcase 降到 18.86 ms/testcase。相比之下，稀疏、小型 TXT 和零列工作负载的加速较低但仍稳定超过 3×。这与第 4 章的瓶颈诊断一致：列数较大的工作负载在 baseline 中制造更多 Jacobi round 边界，因此从 round 级 host/device 往返消除中获得更大收益。

所有工作负载中，baseline 与融合实现报告的总 sweep 数完全一致。例如 `mat_square_medium_*` 均为 512 个总 sweep，`mat_tall_skinny_medium_*` 均为 460 个总 sweep。这一点排除了一个常见误读：当前加速不是因为优化实现减少了迭代次数，而是在相同收敛轨迹下减少了每个 sweep/round 周围的执行开销。

TABLE III
BASELINE 与融合实现的端到端应用耗时对比。时间使用 APP_REPORT.ELAPSED_MS 的 3 次运行均值；扫描数为工作负载内所有 TESTCASE 的总 SWEEP 数。

工作负载	Baseline ms	融合 ms	加速比	Baseline ms/testcase	融合 ms/testcase	总 sweep
mat_grid_small_auto	2280.7	648.2	3.52×	23.76	6.75	684
mat_ill_conditioned_auto	1342.3	291.9	4.60×	33.56	7.30	452
mat_low_rank_auto	1173.5	314.9	3.73×	29.34	7.87	403
mat_sparse_auto	758.9	245.2	3.09×	18.97	6.13	261
mat_square_medium_auto	7108.4	905.1	7.85×	148.09	18.86	512
mat_square_medium_off	7230.8	1069.8	6.76×	150.64	22.29	512
mat_square_medium_on	6913.9	898.2	7.70×	144.04	18.71	512
mat_tall_skinny_medium_auto	1430.9	311.3	4.60×	22.36	4.86	460
mat_tall_skinny_medium_off	1446.6	346.7	4.17×	22.60	5.42	460
mat_tall_skinny_medium_on	1396.4	360.5	3.87×	21.82	5.63	460
mat_zero_columns_auto	819.4	260.8	3.14×	20.48	6.52	257
txt_grid_small_auto	962.7	231.5	4.16×	15.04	3.62	436
txt_ill_conditioned_small_auto	761.6	247.8	3.07×	23.80	7.74	335
txt_sparse_small_auto	862.0	221.5	3.89×	26.94	6.92	239
几何平均加速	–	–	4.36×	–	–	匹配

TABLE IV
融合实现下 QUEUE-CAPACITY 的端到端耗时扫描，单位为 MS。

工作负载	q=1	q=2	q=4	q=8
mat_square_medium_auto	1720.6	992.0	812.3	788.5
mat_sparse_auto	258.4	250.6	229.5	237.8
mat_ill_conditioned_auto	316.4	328.5	325.6	316.1
mat_tall_skinny_medium_auto	307.0	349.4	347.4	333.3

B. 消融实验

本轮归档中最干净的消融实验（ablation study）是融合实现下的线程池容量扫描；它只改变 pipeline 的 `queue-capacity`，用于观察 host-side concurrency 对端到端时间的影响。表 IV 汇总了四个代表性工作负载的一次运行结果。

该消融显示，host 并发在融合后不再呈全局单调收益。对 `mat_square_medium_auto`，从 q=1 提高到 q=8 可把耗时从 1720.6 ms 降到 788.5 ms，说明重工作负载仍能从多个 testcase 的重叠执行中受益。对 `mat_tall_skinny_medium_auto`，q=1 反而最好；对 `mat_ill_conditioned_auto`，四个容量基本持平。Nsight Systems 的并发统计同时显示，容量增大会提高 CUDA Runtime API 活动线程数；例如 `mat_square_medium_auto` 的最大 active thread 从 q=1 的 3 增至 q=8 的 10。因此，这个消融支持一个有限结论：线程池可以改善某些重型 workload 的吞吐，但它不是融合加速的唯一来源，也不是所有 workload 的单调调优旋钮。

其余因素，包括 round 内核融合、round 级 H2D/D2H 拷贝删除、收敛标志粒度变化和 cooperative kernel 启动，被当前实现作为一个结构性优化共同引入；归档数据不能把它们严格分离为互斥补丁。为了避免过度归因，本章只把完整 fused cooperative kernel 作为主处理组，把 API 调用规模变化作为机制证据。

C. 工作负载敏感性

工作负载敏感性主要体现在三组变量：矩阵形状、布局策略和数值收敛压力。

第一，形状敏感性最强。`mat_square_medium_*` 的总 sweep 数为 512，并且每个 sweep 的 round 数由列数放大，因此 baseline 需要支付大量 round 级 launch、memcpy 和 memset 开销。融合后这些 round 边界留在 device 内部，方阵组获得最高加速。高瘦矩阵（tall-skinny matrix）组总 sweep 数接近，为 460，但列数上限只有 64，每个 sweep 内 round 与列对槽更少，所以加速下降到 3.87×–4.60×。

第二，layout transpose 对绝对时间有影响，但不改变主结论。在方阵组中，融合后 `on` 和 `auto` 分别为 898.2 ms 与 905.1 ms，均优于 `off` 的 1069.8 ms；baseline 中 `on` 也略优于 `off`。这说明列连续访问对 medium square 有实际收益。不过，layout 差异的量级明显小于融合带来的 6.76×–7.85× 加速，因此本轮实验的主效应仍是执行粒度变化。

第三，数值困难度会增加 sweep 数，但不是唯一决定因素。`mat_ill_conditioned_auto` 的总 sweep 为 452，加速为 4.60×；`mat_sparse_auto` 的总 sweep 为 261，加速为 3.09×。这说明更多 sweep 会放大 per-sweep 或 per-round 固定成本，但最终收益还受到列数、输入/输出量、layout 和 pipeline 并发共同影响。评价时不能把“病态矩阵更慢”简单等同于

“数值公式更贵”；在当前实现中，控制边界数量同样是关键变量。

D. 性能剖析对比

Nsight Systems 证实端到端加速来自预期的控制面规模下降。10 个 profile 工作负载上，`cudaLaunchKernel` 调用数的几何平均减少为 423.5×，`cudaMemcpy` 调用数的几何平均减少为 65.2×。代表性 case 如 `mat_square_medium_auto`：普通 kernel launch 从 232,692 次降到 180 次，`cudaMemcpy` 从 155,200 次降到 704 次。融合路径新增的 `cudaLaunchCooperativeKernel` 调用为 sweep 级调用；在同一 case 中为 512 次，正好对应总 sweep 数。

CUDA API 累计时间的组成也发生变化。融合前 10 个 profile case 合计中，`cudaLaunchKernel`、`cudaMemcpy` 与 `cudaMemset` 分别占约 43.0%、36.7% 和 17.6%；三者合计几乎支配 CUDA runtime 时间。融合后，`cudaMemcpy` 占约 45.2%，`cudaMalloc/cudaFree` 与 `cudaMallocHost/cudaFreeHost` 合计约 37.4%，而 `cudaLaunchCooperativeKernel` 与 `cudaMemset` 已不再是最大项。这里仍需谨慎：API summed time 是 host-side duration 求和，不是 wall-clock time；它用于定位开销结构，而端到端加速仍以表 III 为准。

Nsight Compute 给出互补证据。baseline 采样 kernel 的 duration 中位数为 6.496 μs，融合后采样 fused kernel 的 duration 中位数为 280.048 μs；这表明 GPU work 粒度显著变粗。SM throughput 中位数从 1.985% 提升到 7.075%，但仍远未接近饱和；DRAM throughput 中位数仍很低。这组数据说明融合确实把大量微小 kernel 合并为更长的设备端执行单元，但融合后的性能上限并非由单个 kernel 的算术峰值或 DRAM 带宽饱和直接决定。

E. 数值精度

本轮 perf 归档能直接支撑的数值行为结论是收敛轨迹一致：14 个 timing 工作负载中，baseline 与融合实现的 `total_sweeps` 完全匹配，且所有 84 条 timing 记录均为 `ok`。由于 fused cooperative kernel 保持相同 pivot 判定和 round-robin 调度，只把 round 内控制转移到 device 侧，这个 sweep 匹配结果是必要的正确性信号：优化没有通过提前停止或少做迭代来换取速度。

为补齐矩阵级正确性检查，本文从 fused timing 归档的原始输出矩阵重新计算重构误差（reconstruction error）和正交性误差（orthogonality error）。对每个 testcase，输出矩阵流按 U、Σ、V 的顺序解释，并与对应输入矩阵 A 重新配对。计算指标为

$$\epsilon_{\mathrm{rec}} = \frac{\|A - U\Sigma V^T\|_F}{\|A\|_F},$$

$$\epsilon_U = \frac{\|U^TU - I\|_F}{\|I\|_F}, \qquad \epsilon_V = \frac{\|V^TV - I\|_F}{\|I\|_F},$$

其中 Σ 使用输出中的奇异值行向量构造。对 rank-deficient 或 zero-column 输入，完整 U 的零奇异方向并不唯一；因此表 VI 报告有效秩（effective rank）子空间上的 $\epsilon_{U,\mathrm{eff}}$，有效列由相对奇异值阈值 10^{-10} 选取。完整 U 指标仍保存在脚本生成的 CSV 中，用作诊断而不作为这些退化 case 的主要正确性判据。

结果显示，融合实现的重构误差在病态 MAT 工作负载上最坏为 2.13×10^{-11}，在其余工作负载上不超过 5.27×10^{-14}；V 的正交性误差最坏为 5.17×10^{-14}。有效 U 子空间的正交性误差最坏为 3.54×10^{-9}，主要来自较大方阵和高瘦矩阵中的归一化列；病态 TXT 与零列工作负载在剔除近零奇异方向后保持在 10^{-9} 量级或更好。因此，本节现在能支持比 sweep 匹配更强的结论：fused cooperative kernel 在当前 double precision 工作负载上没有破坏矩阵级 SVD 关系，剩余异常只出现在数学上不唯一的近零奇异子空间解释中。

VII. 讨论

第 6 章给出的加速结果可以被概括为 kernel 数量和 host/device 往返次数的大幅减少，但这一概括仍不足以解释优化为何成立、何时失效以及后续瓶颈为何发生迁移。更完整的解释应从三个互补层面展开：第一，融合改变了 Jacobi SVD 在 CUDA 上的执行粒度；第二，融合并未改变算法本身的并行性上限；第三，融合使原先被 round 级控制面开销掩盖的次级成本成为可观测瓶颈。因此，本章不再重复端到端表格，而是讨论这些结果对算法实现、系统调优和后续研究的含义。

A. 为何融合在本问题上有效

本工作的核心经验是：GPU 优化并不总是始于单个 kernel 内部。对当前 one-sided Jacobi SVD 而言，baseline 的主要损失来自算法同步边界与 CUDA runtime 边界之间的直接对应。每个 Jacobi round 在数学上确实是一个顺序阶段；但将每个阶段都实现为主机可见的内核发射（kernel launch）、小规模拷贝和同步，会把算法的细粒度控制流放大成系统级开销。

这一点解释了为什么中等方阵组获得最高加速。方阵组列数较大，每个 sweep 内的 round 数更多；baseline 因此制造出更多 round-like 单元，并在每个单元周围支付固定 runtime 成本。融合路径把这些 round 保留在同一个 sweep 级设备端上下文中，使主

TABLE V
代表性 PROFILE CASE 的 CUDA API 调用数变化。

工作负载	Baseline launch	融合普通 launch	Baseline memcpy	融合 memcpy
`mat_grid_small_auto`	56,502	198	37,920	1,068
`mat_ill_conditioned_auto`	43,444	136	29,032	612
`mat_sparse_auto`	24,207	126	16,214	421
`mat_square_medium_auto`	232,692	180	155,200	704
`mat_tall_skinny_medium_auto`	44,268	216	29,624	716
`txt_grid_small_auto`	29,216	128	19,648	692

TABLE VI
融合实现输出矩阵重新计算得到的数值精度。每行汇总该工作负载全部 FUSED TIMING 运行与 TESTCASE；有效秩使用相对阈值 1.00×10^{-10}。

工作负载	类别	case	有效秩	$\max \epsilon_{\mathrm{rec}}$	$\max \epsilon_{U,\mathrm{eff}}$	$\max \epsilon_{V}$
`mat_grid_small_auto`	normal	288	4–64	1.07×10^{-14}	1.72×10^{-9}	1.01×10^{-14}
`mat_ill_conditioned_auto`	ill-conditioned	120	7–8	2.13×10^{-11}	9.50×10^{-10}	1.74×10^{-14}
`mat_low_rank_auto`	low-rank	120	8–64	1.37×10^{-14}	1.70×10^{-9}	1.65×10^{-14}
`mat_sparse_auto`	sparse	120	8–64	1.07×10^{-14}	1.77×10^{-9}	1.05×10^{-14}
`mat_square_medium_auto`	normal	144	32–256	5.26×10^{-14}	3.54×10^{-9}	5.17×10^{-14}
`mat_square_medium_off`	normal	144	32–256	5.26×10^{-14}	3.54×10^{-9}	5.17×10^{-14}
`mat_square_medium_on`	normal	144	32–256	5.26×10^{-14}	3.54×10^{-9}	5.17×10^{-14}
`mat_tall_skinny_medium_auto`	normal	192	8–64	1.04×10^{-14}	1.89×10^{-9}	1.04×10^{-14}
`mat_tall_skinny_medium_off`	normal	192	8–64	1.04×10^{-14}	1.89×10^{-9}	1.04×10^{-14}
`mat_tall_skinny_medium_on`	normal	192	8–64	1.04×10^{-14}	1.89×10^{-9}	1.04×10^{-14}
`mat_zero_columns_auto`	zero-column	120	4–60	1.02×10^{-14}	1.65×10^{-9}	9.41×10^{-15}
`txt_grid_small_auto`	normal	192	4–48	8.01×10^{-15}	1.43×10^{-9}	7.86×10^{-15}
`txt_ill_conditioned_small_auto`	ill-conditioned	96	6	1.20×10^{-12}	5.77×10^{-10}	1.50×10^{-14}
`txt_sparse_small_auto`	sparse	96	8–64	1.10×10^{-14}	1.68×10^{-9}	1.06×10^{-14}

机只观察 sweep 级收敛信号。于是，加速不是来自减少 sweep 数，也不是来自更激进的数值停止条件；第 6 章中 baseline 与 fused 的 total sweep 完全一致，说明收益来自执行结构，而非少做工作。

这个结果也说明，低 GPU 利用率不能直接归因于单个 kernel 的局部实现效率。baseline 的 Nsight Compute 样本显示单个 kernel 只有数微秒，SM throughput 和 DRAM throughput 都较低；在这种情况下，继续优化单个短 kernel 的访存或指令序列，收益通常会受到细粒度启动与同步成本的限制。融合的价值在于把许多微小执行单元合并成较长的设备端工作，使 launch overhead、H2D/D2H 小拷贝和 memset 的频率下降一个到数个数量级。换言之，本文的优化重点不是为各类边界 case 增加局部补丁，而是将 round 级特殊控制路径重构为 sweep 级常规执行路径。

不过，这种有效性依赖于一个特定条件：原实现的控制面开销必须足够大。若输入规模继续扩大到单个 fused kernel 内部计算已经占主导，或者算法改为块 Jacobi（blocked Jacobi）后每次设备端工作本身足够重，那么同样的融合策略可能只带来较小收益。当前结果因此应被理解为“对 round 级 host/device 往返密集型 Jacobi 实现有效”，而不是“所有 GPU SVD 实现都应使用同一融合方式”。

B. 融合后的瓶颈迁移

融合之后，性能问题并未消失，而是发生了瓶颈迁移。第 6 章的 Nsight Systems 结果显示，`cudaLaunchKernel`、`cudaMemcpy` 和 `cudaMemset` 的 round 级调用膨胀被显著压低；与此同时，`cudaMemcpy`、`cudaMalloc/cudaFree` 和 pinned host memory 的分配释放成为 CUDA API summed time 中更显眼的部分。这表明第一层控制面瓶颈被缓解后，资源生命周期和端到端数据移动成本成为新的优化对象。

融合后的剩余成本可以分为四类。第一类是必要数据搬运。程序最终要输出 U、Σ 和 V，因此部分 D2H 拷贝不可避免；但当前 profile 还没有把输入、输出和收敛标志回读拆成独立账本，所以“`cudaMemcpy`

变成最大项”还不能直接推出哪一种 copy 最值得优化。第二类是资源生命周期成本。每个 testcase 反复构造 device buffer 与 pinned staging buffer，在 round 级开销被压低后自然变得突出；workspace 复用（workspace reuse）因此是低风险、高确定性的下一步。第三类是 sweep 级主机交互。当前每个 sweep 仍然需要清零收敛标志、启动 cooperative kernel、回读标志；这已经远好于 round 级回读，但仍不是完全 device-resident。第四类是 fused kernel 内部并行度。Round-robin 调度每个 round 只能处理互不冲突的列对，列数较小时 grid 规模天然偏小；grid-wide synchronization 又要求所有驻留 block 在 round 边界同步，所以设备端吞吐仍难以接近峰值。

这也解释了为什么融合后的 SM throughput 只从很低提升到中低水平，而没有接近饱和。Jacobi 方法的并行性不是无限的：每个 round 内可并行的列对数约为 $n/2$，round 之间存在顺序依赖。Cooperative kernel 把 host 边界换成 device 边界，但没有消除 Jacobi round 本身的同步结构。若希望进一步提高设备利用率，方向更可能是批处理多个小矩阵（batched processing）、块化调度、或针对小列数 case 的专用路径，而不是期待单个矩阵的一个 sweep 自动填满 GPU。

C. 线程池并发的角色

线程池并发（thread-pool concurrency）在本项目中扮演的是吞吐调节器，而不是根因修复机制。Baseline 阶段，多个 host worker 可以交错提交大量短 kernel 和小拷贝，隐藏部分等待，因此 queue-capacity 的适度增加能改善 wall-clock time。但它无法改变单个 testcase 内部每个 round 多次 host/device 往返的执行结构。换言之，pipeline 并发可以改善批处理吞吐量，但不能消除 per-testcase 的结构性控制面开销。

融合后，这个结论变得更具工作负载依赖性。线程池容量扫描显示，`mat_square_medium_auto` 仍能从更大的 in-flight 窗口获益，而 `mat_tall_skinny_medium_auto` 在 q=1 反而最好，病态矩阵 case 基本持平。这说明原先的全局 launch pressure 被削弱后，线程池收益开始受工作负载形状、输出规模、allocator 竞争和 runtime 内部调度影响。因此，queue-capacity 不应被视为单调调优参数，而应作为 workload-aware pipeline 配置的一部分。

更稳妥的工程方向是把 pipeline 背压和 GPU work 并发分离。当前 queue-capacity 同时影响前端提交、future 队列、输出阶段和实际 in-flight testcase 数；它是一个端到端背压参数，而不是纯粹的 CUDA stream 或 GPU worker 数。后续若要系统调优，应至少区分三件事：读入/解析并发、GPU 计算并发、输出写回并发。只有这样，才能判断性能变化来自 GPU overlap，还是来自 CPU 队列、内存分配、文件输出或 runtime 竞争。

D. 算法与实现边界

融合协作内核的第一条边界来自 CUDA cooperative launch 的驻留约束。Grid-wide synchronization 要求同一个 grid 的 block 能够同时驻留；这使 grid 大小受 SM 数量、寄存器、共享内存和线程数共同限制。实现通过 occupancy 查询裁剪 grid，并在条件不满足时回退到 baseline 路径。这种回退很重要，因为优化路径不应成为正确性的前提。我们不应为了一个快路径牺牲可运行性；“不破坏用户空间”在这里翻译成 CUDA 程序就是：设备不支持时，程序仍应给出正确结果。

第二条边界来自算法并行度。One-sided Jacobi SVD 的优势是数值稳定性和正交性表现好，但其 round 结构包含全局顺序。融合把 round 级控制移入设备，并利用 round 内列对互不冲突的性质并行执行；它没有把 Jacobi 旋转变成任意可交换的操作。对于列数很小的输入，round 内并行度不足；对于列数很大的输入，cooperative grid 又可能受驻留上限约束，需要一个 block 串行处理多个列对槽。这两端都限制了单个 fused kernel 的上限。

第三条边界来自数值语义。当前实验支持 fused path 保持矩阵级 SVD 关系：重构误差、V 的正交性误差和有效秩子空间上的 U 正交性误差都处在合理范围内。但浮点计算不承诺逐 bit 相同，特别是在近零奇异值、rank-deficient 输入和零列输入中，奇异子空间本身并不唯一。讨论这些 case 时，正确的问题不是“每个输出列是否与某个参考实现完全一致”，而是“重构、奇异值排序、有效子空间正交性和停止条件是否满足预期”。第 6 章采用有效秩子空间指标，正是为了避免把数学上的非唯一性误判为实现错误。

第四条边界来自接口与输出。本文的端到端评测包含输入读取、layout 转换、SVD 主体、$U/\Sigma/V$ 构造和输出写回。对应用用户而言，这是合理口径；但对 kernel 研究而言，它会把算法主体和 I/O/格式成本混在一起。融合后 `cudaMemcpy` 与分配释放成本变得显眼，说明下一轮优化需要更细的分层计时。否则，我们可能会把输出格式问题误认为 Jacobi kernel 问题，或者把 allocator 问题误认为数学算法问题。

E. 有效性威胁

本研究的内部有效性（internal validity）主要受测量口径影响。Nsight Systems 的 CUDA API summed time 是多线程 API duration 的求和，不等于 wall-clock time；它适合定位开销结构，不适合直接当作端到端耗时。本文在加速比上使用应用报告的 `elapsed_ms`，并用外部 wall-clock median 交叉核对，这是必要的防护。但 profile 本身仍可能扰动调度、

改变缓存状态或影响线程交错，因此 profile 数据应作为机制证据，而不是替代 timing 数据。

构造有效性（construct validity）的威胁在于，本文用 14 个工作负载代表 Jacobi SVD 的输入空间。这些 case 覆盖了方阵、高瘦矩阵、稀疏、低秩、病态和零列场景，但仍不能穷尽真实应用中的矩阵尺寸分布、批大小、输出需求和精度要求。例如，小型 batched SVD、超大单矩阵 SVD 和只需要奇异值的应用，会暴露完全不同的成本结构。当前结论对“需要完整 U, Σ, V 输出的一批中小规模矩阵”最有解释力。

外部有效性（external validity）的威胁来自硬件和软件栈。实验在当前 RTX 3070 Ti Laptop GPU、CUDA 工具链和驱动环境下完成；不同 GPU 的 SM 数量、cooperative launch 支持、内存层次、PCIe/UMA 行为和 runtime 实现都可能改变绝对时间。尤其是 cooperative kernel 的驻留约束和 pinned memory 成本，具有明显平台依赖。因此，本文更强的结论是机制性的：当 round 级 API 调用支配开销时，提升执行粒度会有效；较弱的结论才是具体加速倍数。

结论有效性（conclusion validity）的威胁来自消融不足。当前 fused 实现一次性引入了 round 内核融合、round 级 pairs 拷贝删除、收敛标志粒度变化和 cooperative launch；归档数据能证明完整 fused path 有效，却不能严格分离每个因素的独立贡献。第 6 章中线程池容量扫描是一个较干净的消融，但它只覆盖 pipeline 并发维度。未来若要更精确归因，应构造更多中间版本，例如只删除 pairs H2D、只合并统计和旋转 kernel、只改变收敛标志回读频率，分别测量其贡献。

F. 后续优化方向

最直接的下一步是 workspace 复用。融合后分配释放成本已经从背景噪声变成可见项，而复用 device buffer、pinned host staging buffer、$V/U/\Sigma$ 临时区和收敛标志缓冲，不改变 Jacobi 数值路径，风险低于继续重写 kernel。理想实现应按 worker 或 pipeline 维护容量可增长的工作区，对相同或相近 shape 复用已分配容量，而不是在每个 testcase 上重新分配。

第二步是拆分并优化 copy 路径。需要把 H2D 输入、D2H 输出、sweep flag 回读、layout 转换相关拷贝分别计时。若输出 U, Σ, V 是必须的，则优化重点可能是 staging 和批量写出；若某些应用只需要奇异值或只需要 V，接口层可以提供更轻量的输出模式。对于 sweep flag，可以评估每 K 个 sweep 回读一次、异步回读、或 device-side termination。代价是可能多做 sweep 或引入更复杂的停止协议，因此必须和数值误差、最大 sweep 限制一起验证。

第三步是重新考虑 CUDA Graphs（CUDA 图）与持久内核（persistent kernel）。CUDA Graphs 适合重复提交结构稳定的 GPU work，可降低 launch overhead [8]；但当前 fused path 已经把 round 级 launch 压低，Graphs 的收益可能主要在 sweep 级循环和固定初始化/收尾阶段。Persistent kernel 则可能进一步减少 host 参与，但会带来设备端任务队列、终止条件、资源占用和多 testcase 公平性问题。它们都值得实验，但不应在 workspace 和 copy 拆账之前贸然复杂化系统。

第四步是做 workload-aware 的并发与 kernel 策略。对于中等方阵，较大 queue-capacity 仍有吞吐收益；对于高瘦或较小矩阵，过多并发可能只放大 runtime 与 allocator 竞争。一个更好的策略是根据 m, n、预计 sweep 数、输出规模和 layout 模式选择并发窗口。更进一步，小矩阵可以走 batched cooperative kernel 或非 cooperative 专用路径，大矩阵走当前 fused sweep kernel，高瘦矩阵则重审 block 到列对槽的映射。该方向的目标是使常见工作负载映射到简单、稳定且与其并行度匹配的执行路径。

VIII. 结论与未来工作

本文从性能剖析驱动优化的角度研究 CUDA 上的单边 Jacobi SVD（one-sided Jacobi SVD）。核心结论是：在朴素 round 级映射中，主导瓶颈并非 Jacobi 旋转公式本身，也不是设备端 SM 或 DRAM 吞吐饱和，而是算法同步粒度与 CUDA runtime 执行粒度之间的错配。Baseline 把每个 Jacobi round 表达为主机可见的 kernel launch、H2D/D2H copy、memset 和同步边界，使大量微秒级设备端工作被高频控制面操作包围。Nsight Systems 与 Nsight Compute 的证据共同支持这一判断：中等方阵工作负载产生数十万级 CUDA API 调用，而被采样核心 kernel 的持续时间和硬件吞吐率均较低。

针对这一瓶颈，本文设计并评估了融合协作内核（fused cooperative kernel）。该设计不改变 Jacobi SVD 的数值语义，而是将 round 级列对调度、旋转统计、旋转应用和 round 间同步移动到 sweep 级 cooperative kernel 内部。主机侧因此从观察每个 round 转为观察每个 sweep 的收敛状态。实验结果表明，该执行结构在 14 个工作负载上均取得端到端加速，应用报告耗时的几何平均加速为 4.36×，外部 wall-clock median 的几何平均加速为 4.31×；中等方阵组最高达到 7.85×。同时，baseline 与融合实现的 total sweep 完全一致，融合路径的重构误差和正交性误差也保持在当前 double precision 工作负载的合理范围内。因此，加速主要来自执行粒度重构，而不是减少迭代次数或放宽数值条件。

本文的更一般启示是，GPU 数值算法优化需要同时分析算法同步结构和系统执行边界。线程池并发

可以改善批处理吞吐量，但不能消除单个 testcase 内部的 round 级结构性开销；局部 kernel 调优也难以弥补过细执行粒度带来的 launch 与同步成本。相反，当 profiling 显示控制面开销在设备端吞吐饱和之前已经占据主导时，优先改变 host/device 控制边界通常比微调短 kernel 更有效。

未来工作首先应围绕融合后暴露出的次级瓶颈展开。第一，当前每个 testcase 仍会反复分配 device buffer、pinned host staging buffer 以及 $U/\Sigma/V$ 临时区；worker-local 或 pipeline-local workspace 复用是低风险的下一步。第二，应拆分并量化 H2D 输入、D2H 输出、sweep 收敛标志回读和 layout 转换相关 copy，避免将所有 `cudaMemcpy` 成本归为同一类。第三，可以进一步减少 sweep 级 host 交互，例如评估每 K 个 sweep 回读一次收敛标志、异步 flag copy、CUDA Graphs（CUDA 图）或 device-side termination，但这些方案必须同时验证停止时机、额外 sweep 数和数值误差。

更长期的方向是提高 fused kernel 内部的有效并行度，并使执行策略对 workload 形状敏感。小列数矩阵可探索 batched cooperative kernel 或专用非 cooperative 路径，以避免单矩阵 round 内并行度不足；中等方阵可继续利用 sweep 级融合和适度 pipeline overlap；高瘦矩阵则需要重新评估 block 到列对槽的映射和 queue-capacity 策略。此外，当前结论仍应在更多 GPU 架构、CUDA 版本、矩阵尺寸分布和输出需求上复验。本文给出的结果表明，round 级控制面开销是 CUDA Jacobi SVD 中一个可测量、可解释、可修复的性能问题；后续优化应继续沿着可复现 profile 证据推进，而不是把加速比本身作为唯一目标。

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013, ISBN: 9781421407944.

[2] Z. Drmač and K. Veselić, "New fast and accurate jacobi svd algorithm. i," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 4, pp. 1322–1342, 2008. DOI: 10.1137/050639193.

[3] NVIDIA Corporation, *Cusolver documentation*, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cusolver/ (visited on 05/15/2026).

[4] V. Novaković, "A hierarchically blocked jacobi svd algorithm for single and multiple graphics processing units," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C1–C30, 2015. DOI: 10.1137/140952429.

[5] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, "Batched qr and svd algorithms on gpus with applications in hierarchical matrix compression," *Parallel Computing*, vol. 74, pp. 19–33, 2018. DOI: 10.1016/j.parco.2017.09.001.

[6] J. Dongarra, M. Gates, A. Haidar, *et al.*, "The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale," *SIAM Review*, vol. 60, no. 4, pp. 808–865, 2018. DOI: 10.1137/17M1117732.

[7] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 51–90, 1958. DOI: 10.1137/0106005.

[8] NVIDIA Corporation, *Cuda c++ programming guide*, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 05/15/2026).

[9] NVIDIA Corporation, *Nvidia nsight systems documentation*, 2026. [Online]. Available: https://docs.nvidia.com/nsight-systems/ (visited on 05/15/2026).

[10] NVIDIA Corporation, *Nvidia nsight compute documentation*, 2026. [Online]. Available: https://docs.nvidia.com/nsight-compute/ (visited on 05/15/2026).

Appendix A
实现细节

本附录记录 `jacobi-svd-cuda-opt` 中与实验结论直接相关的工程事实。双栏版式下不逐项列出源文件和函数名；除少数必要接口外，本文用执行阶段、数据流和同步边界描述实现。这样做牺牲了一点源码定位精度，但换来更稳定的 IEEE 排版和更清楚的实验依赖关系。

A. 源码分层与公共接口

实现分为四层。接口层（interface layer）处理命令行参数、配置打印和 JSON 报告；应用层（application layer）把输入、计算和输出组织成流水线；领域层（domain layer）实现 CUDA Jacobi SVD、设备矩阵、布局转置和错误处理；基础设施层（infrastructure layer）处理 MAT/TXT 文件、内存映射（memory-mapped file）和页锁定主机内存（pinned host memory）。

领域层公共入口接收行主序（row-major）主机矩阵、维度和 SVD 配置，返回 U、Σ、V 以及实际 sweep 数。当前实现要求 $m \geq n$。这不是 SVD 的数学限制，而是当前单边 Jacobi 路径的工程前提；宽矩阵需要由调用方先转置，或走另一条算法路径。

B. 运行配置与实验开关

运行配置控制三类量：数值停止条件、kernel launch 形状，以及布局转置策略。数值停止条件包括相对阈值 ϵ 和最大 sweep 数；launch 形状主要是每个 block 的线程数；布局策略有 auto、on、off 三种。auto 模式只有在列数和元素数都超过阈值时才启用列连续布局。

默认配置和归档实验取值由程序配置打印与运行 manifest 记录，论文不在附录中复制具体命令行字面值。线程数进入 CUDA 前会先限制到实现允许的范围，再按 warp 大小对齐；manifest 需要记录归一化前后的 launch 形状，以便复查实验实际运行配置。

C. 单个 *Testcase* 的生命周期

每个 testcase 独立分配 A、V、U 和 Σ 四类设备缓冲区。A 保存输入矩阵的工作副本，V 初始化为单位矩阵，U 和 Σ 在所有 sweep 结束后由收敛后的 A 构造。若启用布局转置，还会额外分配一份列连续的 A 工作副本。

单例执行路径可以概括为八步：校验输入；分配设备内存；拷入 A；必要时做布局转置；初始化 V；执行 Jacobi sweep；必要时把 A 转回行主序；构造并拷回 U、Σ、V。因此，本文的端到端计时包含分配、拷贝、转置、Jacobi kernel、最终同步和结果返回，而不是孤立 kernel 的设备端耗时。

D. 列对调度

实现使用巡回赛调度（round-robin schedule）生成无冲突列对。若 n 为奇数，则加入 dummy 列，使调度在偶数列集合上运行。每个 sweep 包含 n'-1 个 round，其中 n' 是加入 dummy 后的偶数列数；每个 round 至多包含 $n'/2$ 个列对。

该调度的关键性质是：同一 round 内任意真实列最多出现一次。因此同一 round 的多个列对可以并行处理，而不会对 A 或 V 的同一列产生写写冲突。baseline 在主机端生成调度表，并在每个 round 把列对拷到设备；融合路径在设备端重算列对位置，从而去掉 round 级列对拷贝。

E. Baseline 回退路径

当融合路径不可用时，代码进入 round 级 baseline。每个 round 依次执行列对拷贝、收敛标志清零、列对统计、旋转应用和收敛标志回读。列对统计在 block 内用共享内存归约（shared-memory reduction）计算 a_{pp}、a_{qq} 和 a_{pq}；旋转阶段根据这些统计量更新 A 与 V 的两列。

这条路径的优点是简单，并且不要求 GPU 支持协作启动；缺点是每个 round 都支付一次主机/设备交互和多次 runtime 调度成本。第 II 表中的 API 数量膨胀，正是这种执行结构的直接结果。

F. Fused Cooperative Sweep 路径

优化路径先检查设备是否支持协作启动（cooperative launch），再根据 occupancy 估计可同时驻留的 block 数。由于 cooperative groups（cooperative groups）的 grid 级同步要求所有参与 block 能同时驻留，grid 规模取驻留上限和当前 round 可用列对槽数量的较小值。

融合后，主机在每个 sweep 只做三件事：清零 sweep 级收敛标志，启动一个 cooperative kernel，并回读该 sweep 是否发生过旋转。round 级调度、列对统计、旋转计算、A/V 更新和 round 间同步都移动到同一个设备端执行上下文中。若列对槽多于驻留 block 数，同一个 block 会在一个 round 内串行处理多个互不冲突的列对；round 结束时执行 grid 级同步，保证下一轮开始前所有列更新已经完成。

融合路径没有改变每个列对的数学判定：

$$|a_{pq}| > \epsilon \sqrt{\max(a_{pp}, 0)\max(a_{qq}, 0)}. \tag{12}$$

实现还要求 $|a_{pq}| > 10^{-300}$，以避免极小除数。满足条件时，旋转参数使用稳定的 Rutishauser 型公式：

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}}, \qquad t = \begin{cases} \frac{1}{\tau + \sqrt{1+\tau^2}}, & \tau \geq 0, \\ \frac{-1}{-\tau + \sqrt{1+\tau^2}}, & \tau < 0, \end{cases} \tag{13}$$

$$c = \frac{1}{\sqrt{1+t^2}}, \qquad s = tc. \tag{14}$$

随后对 A 和 V 的两列应用同一个正交旋转。由于融合路径改变了同步位置和浮点归约顺序，本文不要求逐 bit 相同；正确性判断以 sweep 数、收敛标志、重构误差和正交性误差为准。

G. 内存布局与转置路径

默认输入和输出矩阵均为行主序。单边 Jacobi 算法反复访问列，直接路径会产生跨步访问（strided access）。布局转置路径把逻辑矩阵 $A(m \times n)$ 重排为列连续布局，使同一列上的连续行在内存中相邻。核心 kernel 通过统一索引函数选择行主序或列连续布局，避免在计算主体中散落布局分支。

转置 kernel 使用 32×32 共享内存 tile，并在第二维额外加一列，以降低共享内存 bank conflict（bank conflict）风险。实验中的 auto/off/on 三组 layout case 不是三个不同算法，而是同一个算法在不同物理布局下运行；它们用于观察列访问连续性在融合后是否成为可见因素。

H. 内存管理

设备矩阵和临时缓冲区采用 RAII（Resource Acquisition Is Initialization）风格管理：构造或重置时分配，析构时释放。这个选择降低资源泄漏风险，但当前粒度仍是每个 testcase 重新分配主要设备缓冲区。融合后，分配释放成本在 Nsight Systems 中更显眼，并不是因为它变慢了，而是 round 级 launch/copy 调用规模大幅下降后，原先被遮蔽的固定成本成为可观测项。

二进制 MAT 输入采用单游标派发读取器。主线程顺序读取元数据和 payload，并把原始 payload 放入页锁定主机缓冲区；工作线程再解码为矩阵并提交 SVD。页锁定内存可以减少传输路径的不确定性，但它的分配释放也会出现在 profile 中。

I. Pipeline 并发与输出顺序

应用层 pipeline 使用全局线程池提交 testcase 计算任务。MAT 输入由主线程顺序派发，TXT 输入由主线程逐矩阵解析后提交。每个 worker 生成包含 U、$\Sigma(1 \times n)$ 和 V 的输出包。

输出阶段使用 future queue（future queue）和单消费者线程。生产者按 testcase 顺序把 future 放入有界队列，消费者按队列顺序等待并写出。因此，即使后提交的 testcase 先完成，输出也不会越序。队列容量控制主机侧在途任务数量；具体容量作为运行配置字段进入 manifest。

需要区分两种并发。case-level 并发决定不同 benchmark case 是否同时运行；pipeline 内部并发决定同一输入流中多个 testcase 是否可重叠解析、CUDA runtime 调用和输出等待。每次归档运行都应在 manifest 中分别记录这两层并发是否启用，避免把 case 级并发和输入流内部重叠混为同一因素。

J. 输入输出格式

benchmark manifest 将输入划分为 MAT 与 TXT 两类，并记录每组输入的逻辑名称、矩阵维度范围和 testcase 数量。baseline 与 fused 实例复用同一批输入文件；论文不在附录中复制具体路径，以避免归档重排后形成失效引用。MAT 文件是矩阵流。每条记录包含行列元数据和紧随其后的 double payload；元数据使用网络字节序（network byte order），矩阵元素使用 `double`。TXT 格式用空格分隔同一行元素、用换行分隔矩阵行、用空行分隔不同矩阵。

每个输入 testcase 固定输出三张矩阵：$U(m \times n)$、$\Sigma(1 \times n)$ 和 $V(n \times n)$。因此输出侧 D2H 拷贝和文件写入成本与维度相关，不能简单视为 profiling 噪声。融合优化主要减少 Jacobi 迭代内部控制面开销，并不消除最终结果必须返回主机这一事实。

K. Benchmark 与 *Profiling* 实例

实验由统一 benchmark harness 驱动。该 harness 负责构建被测程序、加载实例配置、展开 case 矩阵，并把 timing、Nsight Systems 和 Nsight Compute 结果写入一次运行归档。计时模式使用系统 wall-clock 计时并要求应用输出结构化 report；Nsight Systems 捕获 CUDA、NVTX 和 OS runtime trace；Nsight Compute 使用 kernel filter、kernel 名称过滤或固定 launch index 采样代表性 kernel，避免对大量 launch 做全量 profile。

本文使用的 timing 实例覆盖 14 个 workload，每个 workload 重复 3 次；profile 实例覆盖 10 个代表性 workload，每个 workload 执行一次系统级 trace 和两种 Nsight Compute 采样。baseline 与 fused 实例复用同一批输入 case，差异只在执行结构和被采样 kernel。

每个运行归档应包含 manifest，而不是依赖论文中的硬编码字符串。manifest 至少记录：被测实现版本、输入集合标识、数值停止条件、launch 形状、队列/并发配置、layout-transpose 策略、重复次数、profiler 类型、采样过滤规则和原始输出位置。layout-transpose 的三组策略只对中等方阵与中等高瘦矩阵成组展开；其余工作负载使用自动策略。

L. 回退条件与有效性边界

融合路径可能在以下条件下回退：列数小于 2、设备不支持 cooperative launch、occupancy 查询显示没有可驻留 block，或共享内存/线程数配置导致 cooperative grid 无法形成。回退是显式路径：融合尝试失败后执行 round 级 baseline，而不是静默跳过计算。

当前实现还有几个边界。第一，所有测试矩阵均为 double precision，不能直接外推到 single precision 或 mixed precision。第二，输入要求 $m \geq n$，没有覆盖

宽矩阵。第三，grid 级同步依赖 GPU 对 cooperative launch 的支持与驻留能力。第四，当前 workspace 尚未复用到 worker-local 粒度，所以融合后的分配释放成本是当前实现状态的一部分，而不是 CUDA Jacobi SVD 的理论下限。第五，pipeline 内部可能让多个 testcase 的 CUDA API 在主机侧重叠；因此 Nsight Systems 的 API summed time 只能作为开销归因信号，不能直接等同于 wall-clock time。

Appendix B
完整实验表格

本附录集中放置正文中过大的实验表格。表格分为四组：端到端计时、CUDA Runtime API 规模、Nsight Compute kernel 采样，以及线程池容量扫描。正文只保留支持论证的摘要表和关键观察；这里列出复查异常值、比较重复运行和追踪 profile 证据所需的完整数值。

A. 端到端计时

表 VII 给出 baseline 与 fused 在 14 个 timing 工作负载上的 3 次应用报告耗时、均值和加速比。表 VIII 给出同一批运行的外部 wall-clock 时间和最大 RSS。两张表的口径不同：前者来自程序 JSON report 的 `elapsed_ms`，用于正文加速比；后者来自 `/usr/bin/time`，用于暴露进程级启动、I/O 和内存占用影响。

B. Nsight Systems API 汇总

表 IX 列出 10 个 profile 工作负载的 CUDA API 调用数。表 X 列出对应累计 API 时间。这里的累计时间是 host-side API duration 求和；它适合判断 runtime 开销结构，但不能替代端到端 wall-clock 时间。

C. Nsight Compute Kernel 采样

表 XI 汇总 Nsight Compute 采样 kernel 的 duration、SM throughput 和 DRAM throughput。Baseline 样本主要覆盖 `pair_stats` 与 `apply_rotation`，fused 样本覆盖 `jacobi_sweep`。该表用于支撑正文中的两个判断：baseline kernel 粒度处于微秒级，fused 后 GPU work 粒度显著变粗；同时两者都没有表现出接近 SM 或 DRAM 饱和的形态。

D. 线程池容量扫描

表 XII 给出 fused `queue-capacity` 扫描的完整 timing 记录。表 XIII 给出同一批 Nsight Systems trace 中 CUDA Runtime API 的主机并发统计。两张表合在一起说明一个细节：增大队列容量确实提高了 host-side runtime API 重叠度，但端到端收益仍然强烈依赖 workload 形状。

Appendix C
Benchmark 与性能剖析框架

本文的实验不是由手工命令逐条执行，而是由 `bench.sh` 与 `psrc` 组成的两阶段框架驱动。第一阶段在 `experiments/prof` 下生成原始运行归档，包括 timing 日志、Nsight Systems trace、Nsight Compute report 和程序输出矩阵；第二阶段由 `psrc/extractor` 把这些原始产物转换到 `experiments/perf`，形成正文和附录表格使用的 JSON、CSV 与 Markdown 摘要。这个分层让原始 profiler artifact 保持不可变，同时允许后续反复提取、重算和制表。

A. 框架分层

`bench.sh` 是唯一的实验入口。它解析实例名，加载 `scripts/instances/*.sh` 中的实验实例，创建带时间戳的运行目录，写入 `manifest.env`，按需执行 CMake configure/build，然后逐 case 调用 timing、Nsight Systems 或 Nsight Compute runner。脚本本身保持很薄；通用函数位于 `scripts/bench/core.sh`，实例解析和 case 注册位于 `scripts/bench/instances.sh`，具体 profiler 调用位于 `scripts/bench/runners.sh`。

`psrc` 提供两个 Python 命令。其一是 `generate`，用于生成 MAT/TXT 输入流；其二是 `extract`，用于抽取 `bench.sh` 的结果。项目通过 `pyproject.toml` 暴露这两个 console script（console script）。因此，实验框架的职责边界很明确：Bash 层负责真实运行和 profiler 驱动，Python 层负责可复现输入生成和机器可读结果提取。

B. 输入生成

`psrc/generator` 生成本文使用的矩阵流。MAT 输出使用二进制格式：每个 testcase 先写入大端序（big-endian）的行列元数据，再写入 row-major double payload。TXT 输出使用 ASCII 十进制数，行内空格分隔，矩阵之间空行分隔。两种格式都可被同一个 CUDA executable 读取，因此 MAT/TXT 工作负载差异来自输入路径和矩阵集合，而不是 benchmark harness 的不同。

生成器显式记录形状分布（shape distribution）和数值分布（value distribution）。形状策略包括 grid、uniform、log-uniform、square、tall-skinny 和 wide；数值策略包括 uniform、normal、log-uniform、diagonal-dominant、low-rank、ill-conditioned、sparse 和 zero-columns。随机数由固定 seed 和 testcase 全局索引共同决定：

$$\texttt{seed}_{\text{case}} = \texttt{seed} + 10^6 \cdot \texttt{file\_index} + \texttt{case\_index}.$$

因此，只要生成命令、seed、形状集合和分布参数一致，输入矩阵流即可逐 case 再生成。当前论文实验

TABLE VII
完整端到端计时记录：应用报告（APPLICATION REPORT）的 ELAPSED_MS，单位为 MS。每个工作负载 BASELINE 与 FUSED 各运行 3 次；MEAN 用于正文加速比。

工作负载	B-R1	B-R2	B-R3	B-Mean	F-R1	F-R2	F-R3	F-Mean	加速比
mat_grid_small_auto	3310.9	1758.7	1772.3	2280.7	1237.7	371.1	335.8	648.2	3.52×
mat_ill_conditioned_auto	1349.6	1324.8	1352.5	1342.3	295.8	282.9	297.1	291.9	4.60×
mat_low_rank_auto	1190.4	1208.2	1122.1	1173.5	372.4	296.0	276.2	314.9	3.73×
mat_sparse_auto	777.4	737.3	761.9	758.9	242.4	246.1	247.2	245.2	3.09×
mat_square_medium_auto	6975.9	6885.3	7463.9	7108.4	926.9	925.2	863.2	905.1	7.85×
mat_square_medium_off	7357.7	7262.8	7071.8	7230.8	1087.7	1035.2	1086.6	1069.8	6.76×
mat_square_medium_on	7039.5	6750.9	6951.2	6913.9	945.3	891.9	857.5	898.2	7.70×
mat_tall_skinny_medium_auto	1425.7	1450.1	1416.9	1430.9	313.8	311.0	309.0	311.3	4.60×
mat_tall_skinny_medium_off	1497.6	1447.3	1395.1	1446.6	354.7	343.5	341.8	346.7	4.17×
mat_tall_skinny_medium_on	1420.3	1381.2	1387.7	1396.4	344.1	331.3	406.0	360.5	3.87×
mat_zero_columns_auto	790.2	835.7	832.2	819.4	308.5	237.8	236.0	260.8	3.14×
txt_grid_small_auto	984.6	981.1	922.3	962.7	229.5	228.3	236.6	231.5	4.16×
txt_ill_conditioned_small_auto	766.8	761.8	756.2	761.6	234.7	220.7	288.0	247.8	3.07×
txt_sparse_small_auto	983.8	794.0	808.3	862.0	228.4	224.3	211.9	221.5	3.89×

TABLE VIII
外部计时与内存占用汇总。WALL TIME 来自 /USR/BIN/TIME 的 ELAPSED SECONDS；RSS 为 3 次运行中的最大 RESIDENT SET SIZE。

工作负载	B wall mean	B wall min–max	F wall mean	F wall min–max	Wall 加速	B max RSS KB	F max RSS KB
mat_grid_small_auto	2.710	1.790–4.550	0.670	0.360–1.260	4.04×	103,868	103,584
mat_ill_conditioned_auto	1.363	1.350–1.370	0.330	0.300–0.370	4.13×	111,228	111,868
mat_low_rank_auto	1.197	1.140–1.240	0.340	0.300–0.400	3.52×	107,332	106,796
mat_sparse_auto	0.780	0.760–0.800	0.267	0.260–0.270	2.92×	107,380	106,620
mat_square_medium_auto	7.133	6.910–7.490	0.927	0.880–0.950	7.70×	132,668	132,860
mat_square_medium_off	7.600	7.090–8.430	1.093	1.060–1.110	6.95×	133,464	132,396
mat_square_medium_on	6.933	6.770–7.060	0.920	0.880–0.970	7.54×	132,944	131,680
mat_tall_skinny_medium_auto	1.453	1.440–1.470	0.337	0.330–0.340	4.32×	116,340	116,096
mat_tall_skinny_medium_off	1.470	1.420–1.520	0.370	0.360–0.380	3.97×	116,028	115,836
mat_tall_skinny_medium_on	1.853	1.410–2.710	0.380	0.350–0.430	4.88×	116,472	116,336
mat_zero_columns_auto	0.840	0.810–0.860	0.283	0.260–0.330	2.96×	107,300	107,108
txt_grid_small_auto	0.987	0.940–1.010	0.253	0.250–0.260	3.89×	100,100	100,100
txt_ill_conditioned_small_auto	0.783	0.780–0.790	0.737	0.240–1.720	1.06×	100,744	100,580
txt_sparse_small_auto	0.887	0.820–1.010	0.240	0.230–0.250	3.69×	100,904	101,068

固定使用 `experiments/cases/baseline` 下的输入集合，baseline 与 fused 实例复用同一批文件。

C. 实验实例

一次实验由 instance script 定义，而不是由论文正文拼接命令。每个实例声明四类字段：重复次数 `RUNS`，case 级并发 `CASE_JOBS`，启用模式 `MODES`，以及所有运行共享的 `APP_ARGS`。随后用 `add_case` 注册 workload 名称、输入文件和 case 专属参数。对 profiler 模式，实例还必须用 `case_nsys_args`、`case_ncu_basic_args` 和 `case_ncu_deep_args` 指定每个 case 的 profiler 过滤规则。

本文主要使用五个归档实例。`timing.sh` 与 `fused_timing.sh` 覆盖 14 个 workload，每个 workload 运行 3 次，只启用 timing 模式。`profile.sh` 与 `fused_profile.sh` 覆盖 10 个代表性 workload，每个 workload 执行一次 Nsight Systems trace、一次 basic Nsight Compute 采样和一次 full Nsight Compute 采样。`fused_thread_pool.sh` 扫描 `queue-capacity` 为 1、2、4、8 的四组 fused workload，并同时保留 timing 与 Nsight Systems trace。所有这些实例都设置 `CASE_JOBS=1`，避免不同 benchmark case 的 profiler 输出互相竞争；pipeline 内部的 testcase 并发仍由应用的 `queue-capacity` 控制。

D. Timing 模式

Timing 模式用 `/usr/bin/time` 包裹 CUDA executable，并要求应用打印 JSON report。外部计时记录 elapsed seconds、user seconds、system seconds 和最大 resident set size；应用 report 记录 `elapsed_ms`、`testcase_count`、`emitted_matrix_count`、`total_sweeps`、layout-transpose 配置和 autotune 状态。正文的端到端加速以应用 report 的 `elapsed_ms` 为主，因为它排除了

TABLE IX
完整 Nsight Systems CUDA API 调用数对比。Launch reduction 只比较普通 cudaLaunchKernel；Coop launch 单独列出，因为它对应 fused sweep 级 cooperative kernel。

工作负载	B launch	B memcpy	B memset	F launch	F coop	F memcpy	F memset	Launch red.	Memcpy red.
mat_grid_small_auto	56,502	37,920	18,768	198	684	1,068	684	285.36×	35.51×
mat_ill_conditioned_auto	43,444	29,032	14,436	136	452	612	452	319.44×	47.44×
mat_sparse_auto	24,207	16,214	8,027	126	261	421	261	192.12×	38.51×
mat_square_medium_auto	232,692	155,200	77,504	180	512	704	512	1292.73×	220.45×
mat_square_medium_off	232,608	155,200	77,504	96	512	704	512	2423.00×	220.45×
mat_square_medium_on	232,704	155,200	77,504	192	512	704	512	1212.00×	220.45×
mat_tall_skinny_medium_auto	44,268	29,624	14,684	216	460	716	460	204.94×	41.37×
mat_tall_skinny_medium_off	44,180	29,624	14,684	128	460	716	460	345.16×	41.37×
mat_tall_skinny_medium_on	44,308	29,624	14,684	256	460	716	460	173.08×	41.37×
txt_grid_small_auto	29,216	19,648	9,696	128	436	692	436	228.25×	28.39×

TABLE X
完整 Nsight Systems CUDA API 累计时间对比，单位为秒。Malloc+Free 合并 cudaMalloc/cudaFree/cudaMallocHost/cudaFreeHost。

工作负载	B launch	B memcpy	B memset	B alloc/free	F launch+coop	F memcpy	F memset	F alloc/free
mat_grid_small_auto	2.755	2.300	1.102	0.349	0.088	0.292	0.051	0.407
mat_ill_conditioned_auto	1.613	1.538	0.594	0.252	0.100	0.226	0.028	0.277
mat_sparse_auto	0.883	0.890	0.352	0.228	0.074	0.120	0.023	0.212
mat_square_medium_auto	13.332	10.630	5.548	0.274	0.266	1.317	0.172	0.827
mat_square_medium_off	13.317	11.202	5.818	0.302	0.317	1.537	0.147	1.153
mat_square_medium_on	12.488	10.431	5.059	0.333	0.320	1.418	0.131	0.547
mat_tall_skinny_medium_auto	1.369	1.468	0.505	0.299	0.116	0.247	0.038	0.342
mat_tall_skinny_medium_off	1.423	1.557	0.515	0.259	0.109	0.289	0.026	0.286
mat_tall_skinny_medium_on	1.449	1.548	0.497	0.279	0.090	0.235	0.035	0.288
txt_grid_small_auto	1.569	1.304	0.606	0.525	0.075	0.190	0.031	0.520

部分进程外壳噪声；Appendix B 同时列出外部 wall time，用于检查启动、I/O 和内存占用是否出现异常。

每次 timing 运行都会写出完整输出矩阵流。这个选择成本较高，但它避免了一个常见陷阱：只测 kernel 而不测结果返回。本文的性能结论因此包含最终 U、Σ、V 回传和文件写出成本；数值精度脚本也能从同一批输出重新计算 reconstruction error 与 orthogonality error。

E. Nsight Systems 模式

Nsight Systems 模式使用 `nsys profile` 捕获 CUDA、NVTX 和 OS runtime 事件，并关闭 CPU 采样（`--sample=none`）。每个 case 生成 `.nsys-rep` 与 SQLite trace。该模式用于回答控制面问题：CUDA Runtime API 调用了多少次，调用时间堆在哪些 API 上，多个 host thread 是否同时提交 CUDA work。

`psrc/extractor/nsys.py` 优先读取 SQLite trace；若 SQLite 不存在，则回退到 `.nsys-rep`。提取阶段调用 `nsys stats --format json`，默认导出 `cuda_api_sum`、`cuda_gpu_kern_sum`、`cuda_gpu_mem_time_sum`、`cuda_gpu_mem_size_sum` 和 `osrt_sum`。本文主要使用 `cuda_api_sum` 比较 `cudaLaunchKernel`、`cudaLaunchCooperativeKernel`、`cudaMemcpy`、`cudaMemset` 与 allocator API 的调用规模；thread-pool 消融还直接从 SQLite 统计 CUDA Runtime API 区间重叠度。

F. Nsight Compute 模式

Nsight Compute 模式被拆为 `ncu-basic` 与 `ncu-deep`。前者使用 `--set basic`，后者使用 `--set full`。实例脚本必须为每个 NCU case 提供 kernel filter；如果缺少 `--kernel-name`、`--kernel-id` 或 NVTX 过滤，`bench.sh` 会直接拒绝运行。这个限制是有意的：baseline 会产生数万到数十万次 kernel launch，未过滤的 NCU profile 会把一次实验拖成不可复查的超大报告。

Baseline profile 使用 `pair_stats_kernel` 作为 basic 采样目标，用于观察微小统计 kernel 的 duration、SM throughput 和 DRAM throughput。Fused profile 使用 `jacobi_sweep_kernel` 作为采样目标，用于观察 sweep 级融合后 kernel 粒度和资源使用。提取阶段通过 `ncu --import --csv --page raw --print-units base --print-fp` 导出原始 CSV，并额外写出包含稳定字段的 `metrics.json`，例如 kernel 名称、grid/block 形状、duration、SM/DRAM throughput、寄存器数、共享内存和 replay pass 数。

TABLE XI
NSIGHT COMPUTE 采样 KERNEL 指标汇总。DURATION 使用采样行的 `gpu_time_duration.sum`，单位为 μs；SM 与 DRAM 为峰值持续吞吐百分比。

实现	工作负载	采样 kernel	样本	Duration median	Duration min–max	SM median	DRAM median
Baseline	`mat_grid_small_auto`	`pair_stats`, `apply_rotation`	3	6.464	4.352–6.496	0.410	0.200
Baseline	`mat_ill_conditioned_auto`	`pair_stats`, `apply_rotation`	3	6.528	5.056–6.816	5.190	2.450
Baseline	`mat_sparse_auto`	`pair_stats`, `apply_rotation`	3	6.752	5.216–7.008	8.350	4.400
Baseline	`mat_square_medium_auto`	`pair_stats`, `apply_rotation`	3	6.752	4.704–6.848	3.250	1.320
Baseline	`mat_square_medium_off`	`pair_stats`, `apply_rotation`	3	6.592	4.608–6.592	3.250	0.470
Baseline	`mat_square_medium_on`	`pair_stats`, `apply_rotation`	3	6.464	4.672–6.528	6.960	1.310
Baseline	`mat_tall_skinny_medium_auto`	`pair_stats`, `apply_rotation`	3	6.624	4.928–6.656	0.980	0.450
Baseline	`mat_tall_skinny_medium_off`	`pair_stats`, `apply_rotation`	3	6.848	5.024–6.880	3.880	2.130
Baseline	`mat_tall_skinny_medium_on`	`pair_stats`, `apply_rotation`	3	6.496	4.416–6.560	1.000	0.750
Baseline	`txt_grid_small_auto`	`pair_stats`, `apply_rotation`	3	6.432	4.448–6.432	0.410	0.210
Fused	`mat_grid_small_auto`	`jacobi_sweep`	3	65.952	31.520–66.400	1.580	0.060
Fused	`mat_ill_conditioned_auto`	`jacobi_sweep`	3	279.488	146.976–582.976	7.840	0.040
Fused	`mat_sparse_auto`	`jacobi_sweep`	3	583.040	582.432–611.936	15.750	0.070
Fused	`mat_square_medium_auto`	`jacobi_sweep`	3	571.616	280.608–976.192	14.410	0.040
Fused	`mat_square_medium_off`	`jacobi_sweep`	3	616.096	281.152–1605.792	13.390	0.030
Fused	`mat_square_medium_on`	`jacobi_sweep`	3	271.168	271.168–973.440	6.790	0.030
Fused	`mat_tall_skinny_medium_auto`	`jacobi_sweep`	3	279.264	69.312–582.272	7.850	0.040
Fused	`mat_tall_skinny_medium_off`	`jacobi_sweep`	3	298.016	147.200–661.088	7.360	0.050
Fused	`mat_tall_skinny_medium_on`	`jacobi_sweep`	3	67.232	67.232–582.848	1.840	0.080
Fused	`txt_grid_small_auto`	`jacobi_sweep`	3	31.392	31.392–66.016	0.710	0.110

G. 结果提取与表格来源

`extract` 不重新运行 benchmark。它只读取 `experiments/prof/<instance>-<timestamp>`，并在 `experiments/perf/<instance>-<timestamp>` 下写入规范化结果。每个 perf 目录至少包含五类文件：`manifest.json` 保存运行元数据；`runs.jsonl` 保存逐 run 的完整记录；`summary.csv` 保存适合脚本和表格处理的扁平字段；`summary.json` 保存记录数、状态数和 timing 数值摘要；`report.md` 保存人工快速检查用的索引。

日志解析器会记录每次运行的原始命令、warning、error、生成 artifact、report 路径和 profiler kernel 摘要。Timing 记录从日志中抽取 `/usr/bin/time` 字段和应用 JSON payload。Nsight Systems 记录附加规范化 API 表。Nsight Compute 记录附加原始 CSV 和 selected metrics。若 profiler 报错但 artifact 仍存在，记录状态为 `partial`；若既有错误又无 artifact，则记为 `failed`。正文和 Appendix B 只使用状态为 `ok` 的记录。

H. 可复现性约束

该框架用四条约束捍卫可复现性。第一，输入可追踪：instance script 记录每个 workload 的输入文件、格式和 layout 参数；生成器用固定 seed 生成矩阵流。第二，命令可追踪：每个 `run_N.log` 的第一行保存实际执行命令，而不是只保存摘要。第三，原始 artifact 可追踪：`experiments/prof` 保留 profiler 原始输出，`experiments/perf` 只是派生视图。第四，失败显式化：缺失工具、缺失输入、未过滤 NCU、profiler warning/error 都会进入日志或提取记录，而不是在统计阶段静默消失。

这套框架仍有边界。它不能消除 GPU DVFS、OS 调度、driver 状态和其他进程带来的测量噪声；它也不把 `nsys` 的 API summed time 当作 wall-clock time。本文因此用重复 timing 的均值报告端到端性能，用 Nsight Systems 解释控制面规模，用 Nsight Compute 排除明显的 SM/DRAM 饱和假设，并把完整重复记录和 profiler 表格放在 Appendix B 中供复查。

TABLE XII
融合实现 QUEUE-CAPACITY 扫描的完整 TIMING 记录。APP 为应用报告耗时，WALL 为外部 ELAPSED SECONDS。

工作负载	q	App ms	Wall s	User s	Sys s	RSS KB
mat_ill_conditioned_auto	1	316.4	0.340	0.130	0.120	110,300
mat_ill_conditioned_auto	2	328.5	0.350	0.170	0.090	112,024
mat_ill_conditioned_auto	4	325.6	0.350	0.180	0.080	109,788
mat_ill_conditioned_auto	8	316.1	0.340	0.180	0.080	114,348
mat_sparse_auto	1	258.4	0.280	0.060	0.110	106,940
mat_sparse_auto	2	250.6	0.270	0.100	0.080	107,004
mat_sparse_auto	4	229.5	0.250	0.090	0.080	107,032
mat_sparse_auto	8	237.8	0.260	0.100	0.080	106,392
mat_square_medium_auto	1	1720.6	1.740	0.730	0.140	129,904
mat_square_medium_auto	2	992.0	1.020	0.730	0.150	132,288
mat_square_medium_auto	4	812.3	0.830	0.720	0.170	131,196
mat_square_medium_auto	8	788.5	0.810	0.760	0.110	133,876
mat_tall_skinny_medium_auto	1	307.0	0.330	0.190	0.090	115,408
mat_tall_skinny_medium_auto	2	349.4	0.370	0.190	0.100	115,124
mat_tall_skinny_medium_auto	4	347.4	0.370	0.180	0.120	116,148
mat_tall_skinny_medium_auto	8	333.3	0.350	0.170	0.120	117,484

TABLE XIII
融合实现 QUEUE-CAPACITY 扫描的 CUDA RUNTIME API 主机并发统计。SPAN/TOTAL 均按 NSIGHT SYSTEMS RUNTIME API 事件区间计算。

工作负载	q	Runtime events	Span s	Total s	Max active	Avg active	≥ 4 active	≥ 8 active
mat_ill_conditioned_auto	1	2,229	0.269	0.482	3	1.79	0.0%	0.0%
mat_ill_conditioned_auto	2	2,229	0.286	0.580	4	2.02	22.0%	0.0%
mat_ill_conditioned_auto	4	2,229	0.286	0.843	6	2.95	41.5%	0.0%
mat_ill_conditioned_auto	8	2,229	0.267	1.063	10	3.99	48.0%	21.3%
mat_sparse_auto	1	1,636	0.230	0.389	3	1.69	0.0%	0.0%
mat_sparse_auto	2	1,636	0.237	0.455	4	1.92	25.7%	0.0%
mat_sparse_auto	4	1,636	0.192	0.484	6	2.52	38.6%	0.0%
mat_sparse_auto	8	1,636	0.192	0.708	10	3.68	43.2%	18.7%
mat_square_medium_auto	1	2,617	0.883	2.076	3	2.35	0.0%	0.0%
mat_square_medium_auto	2	2,617	0.865	2.688	4	3.11	60.0%	0.0%
mat_square_medium_auto	4	2,617	0.825	3.427	6	4.15	73.1%	0.0%
mat_square_medium_auto	8	2,617	0.837	4.764	10	5.69	76.6%	27.3%
mat_tall_skinny_medium_auto	1	2,773	0.343	0.591	3	1.72	0.0%	0.0%
mat_tall_skinny_medium_auto	2	2,773	0.308	0.665	4	2.16	27.1%	0.0%
mat_tall_skinny_medium_auto	4	2,773	0.305	0.890	6	2.91	45.7%	0.0%
mat_tall_skinny_medium_auto	8	2,773	0.303	1.184	10	3.91	48.6%	20.3%