Jingfeng

Kernel 与 CUDA 编程



面向 AI 研究的系统学习讲义

Jingfeng · 2026 年 10 月 21 日修订

从正确性，到硬件机制，再到性能证据。

一个 AI 算子的工作并不在写出索引之后结束。读者还需要说明：哪些线程拥有数据，谁能观察到谁的写入，浮点结果允许多大误差，以及一次加速究竟减少了什么代价。本书从这些可以核对的问题出发，逐步走向矩阵乘、归一化、Attention、异步流水与多设备通信。

修订版 · CUDA C++ / SIMT / 数据布局 / AI 算子 / 可复现实验

这本书的工作方式
先写数学与输入契约，再构造参考结果；
先画地址与依赖，再选择 tile 和同步；
先固定测量口径，再讨论快了多少。

版本、范围与证据

本书由 2026 年 9 月的原始讲义全文修订而来，本版技术核验截止于 **2026 年 10 月 21 日**。作者标识沿用 Jingfeng。新版保留 CUDA C++、SIMT、内存布局、并行模式、AI 算子和多 GPU 的主线，重写存在前提缺失的算法与 API 示例，并把独立算例、源资料与验证记录纳入同一个项目。

本书主要以可冻结的 CUDA 13.0 文档说明接口语义，必要时对照旧版文档和当前官方工具资料。这里的版本是阅读基准，并非宣称已经在 CUDA 13.0 或任何一款 GPU 上完成性能认证。工具链、驱动、架构目标、库的数学模式需要在读者的机器上重新记录。动态网页均保留了本次读取的地址、日期及摘要或页面快照。

本次修订发生在 macOS ARM64 主机，环境中没有 NVIDIA GPU、CUDA Toolkit 的 `nvcc` 或 Nsight。已运行的实验是 CPU 数学与边界检查；本书没有给出据此推断的 GPU 吞吐、延迟或加速比。随书 CUDA 程序具有完整主机入口和检查协议，GPU 编译、设备执行、Compute Sanitizer 及 profile 项目保留为**待验证**。这种状态是实测范围的说明，不能把“数学验证通过”读成“设备实现已经通过”。

正文中的短代码服务于推导，标为伪代码的段落不能直接编译；`examples/` 下的独立程序才是编译实验入口。格式化示例不包含外部论文的实验数字。公式中的理想流量与手算上界也不是性能观测。

前置知识与阅读路线

读者需要知道数组、循环、函数、矩阵乘和基本浮点数概念。第二章会补齐指针、引用、模板参数、整数提升和 C++ 资源生命周期中与本书直接相关的部分；无需预先掌握复杂模板元编程。

阅读阶段	核心问题	可检验能力
第 1–3 章	谁执行、谁协作、谁能看见写入？	写出边界安全的 SAXPY，解释一个无效 collective。
第 4–7 章	字节如何移动、瓶颈如何证实？	从地址推导访存，区分理想流量与实际流量。
第 8–12 章	算子的数学怎样变成数据流？	推导分块 GEMM 和在线 Softmax，并画出 tile 生命周期。
第 13–15 章	怎样使错误与性能可复核？	保存参考、原始日志、依赖图及同条件的对照实验。

第一次阅读可按上述顺序，在 SAXPY、转置、归约和 GEMM 四个算例处停下来。有框架经验的读者可以先读第十一章，但遇到 stream 或 mask 时应回到前文核对其契约。每章练习和提示用于检查理解，不要求先做完所有架构特化才能继续阅读。

这本书聚焦单算子和运行时机制。大模型架构及推理系统的更广背景，可另读同书系的《大语言模型原理与推理系统》；集群容量规划及服务基础设施，可另读《大模型推理加速与 AI 基础设施》。这些主题指引不代替本书的先修说明，也不使用尚未固定的跨书章节编号。

目录

版本、范围与证据 i

前置知识与阅读路线 ii

第一部分　先证明正确：从线程到协作 1

第一章　从数学任务到 GPU 执行层级 2

1.1 先界定计算，再选择并行方式 . . . 2
1.2 线程、线程块与硬件的关系 . . . 2
1.3 索引是地址证明的一部分 . . . 3
1.4 Grid-stride loop 与空输入 . . . 3
1.5 资源、驻留与 occupancy . . . 4
1.6 同步边界与本章验收 . . . 4

第二章　CUDA C++ 最小语言、编译与运行契约 5

2.1 读懂 kernel 所需的 C++ . . . 5
2.2 限定符与 launch 参数 . . . 5
2.3 从 .cu 到可执行文件 . . . 6
2.4 最小完整程序的生命周期 . . . 6
2.5 错误的出现位置与同步边界 . . . 7
2.6 共享内存与模板的契约 . . . 7
2.7 独立参考、容差与本章练习 . . . 7

第三章　SIMT、参与集合与内存顺序 9

3.1 SIMT 不提供逐行锁步承诺 . . . 9
3.2 Mask 是参与契约，不是自动补零器 . . . 9
3.3 屏障先保证匹配，再讨论数据 . . . 10
3.4 Barrier、fence 与 atomic 的不同职责 . . . 10
3.5 用发布与获取解释通信 . . . 11
3.6 浮点正确性与确定性是两件事 . . . 11
3.7 练习与诊断提示 . . . 12

第二部分　解释代价：内存、并行模式与测量 13

第四章　从地址映射理解内存与矩阵转置 14

4.1 存储位置、作用域与生命周期 . . . 14
4.2 先列出一个 warp 的地址 . . . 14
4.3 布局、行间距与子视图 . . . 15
4.4 用 bank 公式解释 padding . . . 15
4.5 矩阵转置：读写不能只优化一侧 . . . 15
4.6 如何验证布局收益 . . . 16
4.7 主机传输和分配也有生命周期 . . . 16

第五章 Map、归约、Scan 与不规则访问 18
5.1 Map：独立性比“一元素一线程”更基本 . . . 18
5.2 归约：把一条依赖链变为树 . . . 18
5.3 归约的代价与准确性选择 . . . 19
5.4 Scan：每个元素都需要前面的信息 . . . 19
5.5 一个可追踪的 block scan . . . 20
5.6 直方图与热点分布 . . . 20
5.7 邻域计算与 Scatter/Gather . . . 21
5.8 练习与验收 . . . 21

第六章 Roofline、延迟隐藏与可复现测量 22
6.1 先算算术强度，再猜瓶颈 . . . 22
6.2 延迟隐藏不是“把 occupancy 调到最高” . . . 22
6.3 计时协议：先明确时间边界 . . . 22
6.4 正确性、统计和单位一起记录 . . . 23
6.5 从时间线到单 kernel 指标 . . . 23
6.6 第六章检查清单与练习 . . . 23

第七章 优化方法：把收益假设变成实验 25
7.1 按证据排序的优化流程 . . . 25
7.2 常见方法及其收益假设 . . . 25
7.3 编译器提示不是免费的性能开关 . . . 26
7.4 融合与重排：少一次往返仍需付出代价 . . . 26
7.5 第七章检查清单与练习 . . . 26

第三部分 构造算子：矩阵、归一化与运行时 27

第八章 GEMM：从数据复用到 Tensor Core 28
8.1 先把矩阵乘法的契约写完整 . . . 28
8.2 共享内存分块：同步也是算法的一部分 . . . 29
8.3 Register tiling：把共享读取换成线程内复用 . . . 29
8.4 WMMA：一个 warp 共同拥有一个矩阵片段 . . . 30

8.5 精度格式改变了参考问题 31
8.6 继续优化之前，先写收益与代价 31
8.7 渐进实验与解题提示 31

第九章 训练与推理算子：稳定性、融合与在线计算 33
9.1 Softmax：稳定公式需要有效元素契约 33
9.2 在线归一化：合并的是带参考尺度的状态 34
9.3 LayerNorm、Welford 与 RMSNorm 34
9.4 在线 Attention：分母和分子一起换尺度 35
9.5 Prefill 与 Decode：同一公式的两种数据流 36
9.6 反向传播与重算：保存的是足够信息 36
9.7 量化：表示、scale 与累加是一个算法 37
9.8 稀疏与 MoE：零值不是自动省下的工作 37
9.9 练习：让边界输入参与设计 38

第十章 异步执行：把依赖、生命周期与时间线对齐 39
10.1 三个时刻：提交、开始、完成 39
10.2 Event 是依赖边，不只是秒表 39
10.3 资源生命期：什么时候才能复用地址 40
10.4 双缓冲：两个槽位是一张状态机 40
10.5 CUDA Graphs：复用的是工作图与提交结构 41
10.6 结构化协作：CTA、cluster 与 cooperative grid 42
10.7 练习：从故障时序倒推缺失的边 42

第十一章 PyTorch 扩展与 Triton：从 kernel 到框架 43
11.1 先定位调用链和缺口 43
11.2 C++/CUDA extension 的最小契约 43
11.3 Triton 的块级视角 44
11.4 何时使用更高层工具 44
11.5 框架集成的易错点 44
11.6 第十一章检查清单与练习 45

第十二章 现代 GPU 架构与可移植特化 46
12.1 先查 capability，再写 fast path 46
12.2 Ampere、Hopper 与 Blackwell 的差异 46
12.3 异步搬运的完成语义 46
12.4 PTX、SASS 与生成代码核验 47
12.5 第十二章检查清单与练习 47

第四部分　留下证据：调试、通信与研究实验 **48**

第十三章　调试、工具链与可复核证据 **49**

13.1 先定义算子的输入契约 49
13.2 把错误绑定到具体阶段 49
13.3 一条有先后关系的工具链 50
13.4 Nsight 任务应从问题开始 50
13.5 一次运行应留下什么 51
13.6 本章练习与提示 51

第十四章　多 GPU：依赖、通信与计算重叠 **52**

14.1 先确认地址、设备和可用路径 52
14.2 集合通信的正确性先于算法选择 52
14.3 一个 bucket 的依赖关系 53
14.4 为什么已经异步仍然没有收益 53
14.5 怎样测得一个可信的多 GPU 结论 54
14.6 本章练习与提示 54

第十五章　实验路线：从正确结果到优化解释 **55**

15.1 把路线划成三个可验收阶段 55
15.2 随书代码可以怎样开始 55
15.3 渐进 GEMM 的公平对照 56
15.4 本次实际执行的证据与边界 56
15.5 一页报告的内容 57
15.6 本章练习与提示 57

附录 A　接口速查：先核对契约 **58**

A.1 线程、地址与资源 58
A.2 同步与内存顺序 58
A.3 队列、内存与错误检查 59
A.4 从文档查到可运行环境 59

附录 B　符号、单位与手算检查 **60**

B.1 统一符号 60
B.2 用四个算子检查流量口径 60
B.3 把数值契约写成判定式 60
B.4 术语不互相替代 61

附录 C　阅读资料与证据分级 **62**

C.1 原始资料分别回答什么 62
C.2 从数学到算法论文 62

C.3 工具证据的读取顺序 62
C.4 如何继续核验 63

参考文献与原始技术资料 **64**

第一部分

先证明正确：从线程到协作

第一章

从数学任务到 GPU 执行层级

本章的学习目标

本章建立一个可用来推理正确性的执行模型。读完后，应能把长度为零、非整齐长度、带行间距的二维数组映射到线程，说明哪些线程可以协作，并在提交 kernel 前检查索引、覆盖范围和资源限制。性能分析将在第六章加入；此处先保证每个输出由正确的线程产生。

1.1 先界定计算，再选择并行方式

GPU kernel 是在设备上执行的一段函数。本书中的 kernel 不是操作系统内核。主机程序负责分配内存、建立依赖并提交工作，设备程序负责并行计算；函数返回到主机不等于设备计算已经结束。

以 SAXPY 为起点：对 $0 \le i < n$，计算

$$y_i \leftarrow a x_i + y_i.$$

它可以按元素并行，因为不同输出地址互异，且每个输出只依赖输入的同一个位置。这一结论包含隐含条件：x 与 y 指向有效数组，两个指针没有违反实现所声明的别名约束，数组没有被另一个并发任务修改。若计算改为 $y_i \leftarrow x_{i-1} + x_i$，原地覆盖 $x = y$ 就可能让线程读到其他线程已经改写的值。数学表达式看似相近，正确的并行结构却不同。

开始写 kernel 前，用一份短契约回答四件事：输入 shape、dtype 和 stride；输出和输入是否允许重叠；空输入及特殊值的语义；允许多大的数值误差。优化是保持这份契约的实现变换。框架已有高性能算子时应先测量库版本；自写 kernel 的价值常来自特殊布局、融合或新算法，不能以“用了 CUDA”作为性能证据。

总耗时还包含传输和调度。若一个目标算子占原程序时间的比例为 f，它被加速 s 倍，在其余时间不变时，整体加速上界为

$$S = \frac{1}{(1 - f) + f/s}.$$

例如 $f = 0.05$、$s \to \infty$，上界仅约为 1.053。这是手算模型，说明应先定位真实瓶颈；它不是任何 GPU 的实测成绩。

1.2 线程、线程块与硬件的关系

CUDA 把一次提交组织为 grid（网格），grid 由 block（线程块，文献也称 CTA）构成，block 包含 thread。现代 NVIDIA CUDA GPU 把线程以 32 个 lane 组成 warp；本文在使用 warp 原语处会明确 32-lane 前提。软件中的一个线程不是永久绑定的一个物理算术单元。一个 SM（流式多处理器）可以保有多个 warp 的寄存器状态，由调度器选择可执行的 warp 发射指令。[1]

对象	保证或用途	不提供的保证
thread	自己的坐标、寄存器与局部执行状态	与相邻线程按源代码逐行同步
warp	lane 集合，可用显式 warp 协作原语	发散路径上的隐式共享内存同步
block / CTA	共享内存及块内屏障；一个 block 在一个 SM 上执行	与其他 block 的执行先后顺序
grid	一次 kernel 的全部 block	普通 launch 下全部 block 同时驻留
stream	已提交操作的依赖顺序	不同 stream 自动互相等待或必然并发

普通 kernel 中，各 block 必须能以任意顺序执行。不能让所有 block 自旋等待一个尚未得到调度的生产者 block：已驻留的消费者可能占满资源，使生产者无法开始。最容易证明正确的全局分阶段算法，是让阶段一和阶段二成为同一 stream 中的两个 kernel。协作 launch、线程块 cluster 等有特定支持与驻留条件，见第十章；它们不能由普通 block 屏障替代。

1.3 索引是地址证明的一部分

一维编号的数学式是

$$i = b_x T_x + t_x,$$

其中 b_x、T_x、t_x 分别对应 `blockIdx.x`、`blockDim.x`、`threadIdx.x`。若 $n = 1000$、每块 256 线程，需要 4 个 block，共 1024 线程。编号 1000 至 1023 的线程必须不访问数组。

二维行优先数组使用

$$r = b_y T_y + t_y, \qquad c = b_x T_x + t_x, \qquad \text{offset} = r\,\text{ld} + c.$$

ld 是以元素计的行间距，必须至少覆盖一行的逻辑宽度；它不必等于列数。切片、padding 和子矩阵都会产生非紧密布局。`cudaMallocPitch` 返回的 pitch 则以字节计，两者必须先换算。

```
size_t r = size_t(blockIdx.y) * blockDim.y + threadIdx.y;
size_t c = size_t(blockIdx.x) * blockDim.x + threadIdx.x;
if (r < rows && c < cols) {
  size_t off = r * ld + c;
  out[off] = in[off];
}
```

把最终结果赋给 64 位变量还不够：`size_t(blockIdx.y * blockDim.y)` 会先用原操作数的类型乘法，再转换，无法挽救已经发生的溢出。上例在乘法前转换。主机端还需检查 rows ld 与字节乘法是否能被 `size_t` 表示，并满足设备的 grid 维度上限。大索引正确不等于 launch 维度合法；超过二维 grid 范围时，可把 tile 编号线性化后再恢复行列。

多维 block 的线性线程编号按 x 最快变化：

$$t = t_x + T_x(t_y + T_y t_z).$$

warp 按此顺序组成，不能把 `threadIdx.x & 31` 当成任意二维 block 的 lane 编号。例如 $T_x = 16$ 时，一个 warp 横跨两行。后文一维归约代码会明确限定一维 block。

1.4 Grid-stride loop 与空输入

令总启动线程数为 GT，线程 t 处理

$$t,\ t + GT,\ t + 2GT, \ldots < n.$$

不同线程对应不同的模 GT 余数，因此无重复且覆盖所有元素。固定 grid 也能处理长数组，这让调度规模与问题长度分离。循环本身不自动提高吞吐：其收益取决于资源、访存和每线程工作量。

```
__global__ void saxpy(const float* __restrict__ x,
                      float* __restrict__ y, float a, size_t n) {
  size_t i = size_t(blockIdx.x) * blockDim.x + threadIdx.x;
  size_t step = size_t(blockDim.x) * gridDim.x;
  for (; i < n; i += step) y[i] = fmaf(a, x[i], y[i]);
}
```

该循环假定所有有效地址与步进可用 `size_t` 表示，并且 n 不接近整数类型的极限；实际合法分配的浮点数组满足更强的字节容量约束。`__restrict__` 还要求本例的输入和输出不重叠。需要允许 $x = y$ 时，应移除不适用的承诺，并重新审查更一般的重叠情况。

主机端对 $n = 0$ 直接返回，既不分配零长度缓冲区，也不提交 grid 为零的 kernel。向上取整可写成 $n/T + [n \bmod T \neq 0]$，避免 $n + T - 1$ 的整数加法溢出。使用 grid-stride loop 后，可依据设备和测量结果为 grid 设上限，不应把某个“每 SM 几个 block”的经验数写成正确性要求。

1.5 资源、驻留与 occupancy

若每块有 T_B 个线程、每线程使用 R_t 个寄存器、每块使用 S_B 字节共享内存，忽略硬件分配粒度时，同时驻留块数受到以下约束：

$$B_{\text{res}} \le \min\left(\left\lfloor \frac{T_{\text{SM}}}{T_B} \right\rfloor, \left\lfloor \frac{R_{\text{SM}}}{R_t T_B} \right\rfloor, B_{\text{SM,max}}\right).$$

当 $S_B > 0$ 时还需加入 $\lfloor S_{\text{SM}}/S_B \rfloor$。实际寄存器和共享内存按硬件粒度分配，还有 warp 数、每块线程数和架构限制，故手算只给方向。Occupancy 是驻留 warp 数除以硬件允许的最大驻留 warp 数，不是正在执行指令的 lane 比例，也不是利用率。[2]

例如一个教学模型有每 SM 65536 个 32 位寄存器，每块 256 线程。每线程 64 个寄存器对应每块 16384 个寄存器，单看寄存器最多 4 块；改为 65 个后，简单除法最多 3 块，实际还可能因分配粒度进一步变化。模型只说明资源阈值可能带来不连续的驻留变化，不代表任一具体芯片的完整配置。

多驻留一些 warp 可以提供更多可调度工作，隐藏依赖或访存等待；但增大寄存器复用、减少重复加载，即使降低 occupancy，也可能减少总时间。第六章将把驻留机会和真实执行证据分开。

1.6 同步边界与本章验收

`__syncthreads()` 只作用于同一 block；`__syncwarp(mask)` 作用于所指定的 warp 参与集合。事件表达 stream 之间的依赖，主机同步让 CPU 等待设备完成。不同层次的“等待”不能互换，内存顺序还需要单独论证，见第三章。

- 对 $n = 0, 1, 31, 32, 33, 255, 256, 257, 1003$，能手算提交规模与最后一个有效索引。
- 给定 rows、cols、ld 和元素大小，能检查最大访问地址及整数中间值。
- 能给出一个 block 任意调度仍正确的方案，并指出一个跨 block 自旋方案的风险。
- 能解释较低 occupancy 为什么可能更快，并列出尚需测量的证据。

练习与提示。（1）给 17×35、$\text{ld} = 40$ 的矩阵设计二维映射，写出右下角元素偏移，并确认 padding 不被修改。提示：偏移是 $16 \cdot 40 + 34$，需要区分逻辑元素和分配空间。（2）证明 grid-stride loop 无重复。提示：假设两线程产生同一个编号，对总线程数取模。（3）比较一维 block 256 和二维 block 16×16 的前两个 warp。提示：先线性化线程编号，再讨论邻接访存，不能只看线程总数相同。

第二章

CUDA C++ 最小语言、编译与运行契约

本章的学习目标

本章的验收产物是一份能独立编译的 SAXPY 程序：主机创建输入和设备缓冲区，设备计算，主机用独立参考检查结果。要求能解释每个指针属于哪里、哪一步可能异步、错误在何处被发现，而不是只记住 launch 语法。

2.1 读懂 kernel 所需的 C++

C++ 指针存储地址，`p[i]` 等价于对 `p+i` 解引用，地址增量是 $i\,\mathrm{sizeof}(T)$ 字节。`const float* p` 禁止通过该指针改写元素；它不保证其他别名或设备不会修改内存。`size_t` 是无符号大小类型，`int64_t` 是固定 64 位有符号整数；混合有符号与无符号比较可能把负数转换为很大的正数。形状应先验证非负，再转换为大小类型。

`std::vector<float>` 管理 CPU 上的一段连续存储；它的 `data()` 返回主机地址，不能因为传入 kernel 参数就成为合法的设备数组。`cudaMalloc` 返回设备地址，一般也不能在 CPU 直接解引用。Unified Memory 与已注册的映射内存另有访问条件，不应模糊最初的主机/设备区分。

模板 `template<int B>` 让 B 成为编译期常量，`constexpr` 表达可在编译期求值的常量，`static_assert` 检查编译期契约。它们能帮助展开和资源规划，但每组模板参数也可能产生新的二进制实例。C++ 引用是已有对象的别名；主机函数的引用参数、容器和异常不等于可直接传给 GPU 的复杂对象。最初的接口采用标量、尺寸和设备指针即可。

```
template<int B>
constexpr bool supported_block() { return B >= 32 && B % 32 == 0; }
static_assert(supported_block<256>(), "whole warps required");
size_t bytes = n * sizeof(float);  // check overflow before multiplication
std::vector<float> host(n);
float* device = nullptr;
```

2.2 限定符与 launch 参数

语法	含义和约束
`__global__`	设备执行的 kernel；本书从主机提交，返回类型为 void。
`__device__`	从设备调用的函数。
`__host__`	主机函数；普通函数默认属于主机端。
`__host__ __device__`	分别生成主机和设备版本；所调用函数在相应侧必须可用。
`__shared__`	每个 block 独立的共享存储，不是全 grid 的数组。
`__constant__`	设备常量空间；warp 读取同一地址时有广播价值。
`__restrict__`	NVIDIA 支持的无别名承诺，错误承诺会破坏编译器优化所依据的语义。

```
kernel<<<grid, block, dynamic_shared_bytes, stream>>>(arguments);
```

前两个参数是 block 数和每块线程维度；第三个参数是每块动态共享内存字节数，第四个参数选择 stream。`dim3` 的缺省 y, z 为 1。设备内建坐标有 x, y, z 分量，而 `warpSize` 是标量，不能把它误读为三维坐标。

2.3 从 .cu 到可执行文件

`nvcc` 是编译驱动：把主机代码交给兼容的主机 C++ 编译器，设备代码可生成 PTX 虚拟指令或目标架构机器码。`ptxas` 把 PTX 汇编为目标设备代码；驱动在可用时加载匹配机器码，否则可能对兼容 PTX 做即时编译。PTX 向前兼容有工具链和目标约束，不保证任意新 PTX 都能由旧驱动识别。[3, 4]

```
nvcc --version
nvcc -std=c++17 -O3 -lineinfo -Xptxas=-v \
  -arch=sm_80 basic_kernels.cu -o basic_kernels
```

`sm_80` 是示例目标，不是本书所有机器的默认值；应由实际 GPU 的 compute capability 和已安装 Toolkit 支持范围选择。`-lineinfo` 便于 profiler 关联源码，`-Xptxas=-v` 报告资源，`-G` 是设备调试构建，会改变优化与性能，不用于发布性能数字。对需要分发的程序，使用多组 `-gencode` 打包目标机器码及明确选择的 PTX fallback；本书第十二章讨论架构特化和版本矩阵。

记录编译器、主机编译器、驱动和 GPU 是必要步骤。`nvidia-smi` 显示的 CUDA 版本代表驱动所支持的能力信息，并不直接等于当前 `nvcc` 或框架捆绑运行库的版本。在没有 NVIDIA GPU 的主机上可阅读、做 CPU 数学验证；是否可以离线编译还取决于 Toolkit 与目标平台。

2.4 最小完整程序的生命周期

完整示例随书放在 `manuscript/examples/`。阅读时按“输入、分配、传输、提交、回传、验证、释放”的顺序检查。关键 host 包装可以写为：

```
void launch_saxpy(const float* x, float* y, float a, size_t n,
                  int sm_count, cudaStream_t stream) {
  if (n == 0) return;
  constexpr unsigned B = 256;
  size_t need = n / B + (n % B != 0);
  size_t cap = size_t(sm_count) * 8; // tuning choice, not a guarantee
  unsigned G = unsigned(std::min(need, cap));
  saxpy<<<G, B, 0, stream>>>(x, y, a, n);
  CUDA_CHECK(cudaGetLastError());
}
```

这里 `sm_count` 必须来自成功的设备属性查询，并在接口中保证大于零；grid 上限还需满足设备限制。`CUDA_CHECK` 是主机端错误处理宏，定义如下。教学程序遇错立即退出；可复用库应将错误传给调用者并用资源封装保证清理。

```
#define CUDA_CHECK(expr) do {                                  \
  cudaError_t e = (expr);                                      \
  if (e != cudaSuccess) {                                      \
    std::fprintf(stderr, "%s:%d: %s\n", __FILE__, __LINE__,    \
                 cudaGetErrorString(e));                       \
    std::exit(EXIT_FAILURE);                                   \
  }                                                            \
} while (0)
```

主机程序包含 `cuda_runtime.h`、`algorithm`、`cstddef`、`cstdio`、`cstdlib`、`vector` 等对应头文件；设备核使用 `fmaf` 时包含适当数学声明。分配前检查

$$n \le \frac{\max(\texttt{size_t})}{\operatorname{sizeof}(\texttt{float})}.$$

分配成功只证明取得了容量，不证明 shape、stride 和数据依赖正确。释放发生在所有使用该内存的异步任务完成之后。

2.5 错误的出现位置与同步边界

launch 配置不合法常在提交后立刻被 `cudaGetLastError()` 或 `cudaPeekAtLastError()` 发现；非法地址等执行错误通常在后续同步或运行时调用中报告。`Get` 读取并清除当前 host 线程的运行时 last-error 状态，`Peek` 读取但不清除。它们都不能证明 kernel 已经成功执行，也不是同步操作。先前异步任务的错误可能在较晚调用中暴露，因此应记录最早观测到的错误和相关 stream。[1]

调试路径应在每次提交后检查 launch，并在明确边界检查 `cudaStreamSynchronize(stream)` 的返回值。用 `cudaDeviceSynchronize()` 排错很直接，但它等待设备上此前工作，可能掩盖依赖缺失，也会改变并发。生产库不应为方便验证每个算子都做全设备同步；它应遵循调用者的 stream，在应用选择的边界发现异步错误。进一步使用 Compute Sanitizer，见第十三章。

适用条件与易错点

“异步”描述的是主机返回和设备执行之间的关系，不表示所有操作都能重叠。`cudaMemcpyAsync` 对 pageable 主机内存可能发生暂存或阻塞；pinned 内存、可用复制引擎、独立 stream 和正确依赖只是形成重叠的条件，最终需要时间线证据。

2.6 共享内存与模板的契约

下例刻意只展示动态共享内存和屏障，不把它当作缩放算子的性能优化。缩放本来无需共享内存；引入共享只增加资源和同步。

```
template<int B>
__global__ void staged_scale(float* x, size_t n) {
  extern __shared__ float tmp[];
  size_t i = size_t(blockIdx.x) * B + threadIdx.x;
  tmp[threadIdx.x] = i < n ? x[i] : 0.0f;
  __syncthreads();
  if (i < n) x[i] = 2.0f * tmp[threadIdx.x];
}
// Contract: one-dimensional block.x == B; dynamic bytes >= B*sizeof(float).
```

所有线程都到达屏障，只有数组读写由边界条件保护。若模板参数 B 与实际 block 大小不同，索引覆盖和分配容量就可能错位。多个动态数组共享一段 `extern __shared__` 存储时，必须逐段计算大小和对齐，不能只把前一数组元素数当作后一数组的字节偏移。

2.7 独立参考、容差与本章练习

SAXPY 的 CPU 参考应保留输入的初始副本，并在 double 中计算 $ax_i + y_i$。GPU 使用 `fmaf`，乘加只舍入一次；CPU 若分开执行两个 float 运算，可能与之有最后几位差异。优先比较

$$|y_i - y_i^{\mathrm{ref}}| \le \mathrm{atol} + \mathrm{rtol}|y_i^{\mathrm{ref}}|,$$

同时拒绝未约定的 NaN/Inf。容差应由数值规模和算法推导，不应无限放宽直到测试通过。随机测试使用固定种子，还应加入全零、正负交替、极大/极小量级及边界尺寸。对空输入，验收内容是无非法分配、无非法 launch、无访问；不是测一个零元素 kernel 的带宽。

练习与提示。（1）写一个输入长度为零仍成功返回的主机测试，并确认不调用 launch。提示：边界语义属于 host 和 device 两侧。（2）去掉 `__restrict__` 后允许 $x = y$，给出 $y \leftarrow (a+1)y$ 的参考；再解释相差一个元素的重叠为什么不能直接类推。（3）故意把动态共享内存少分配一个元素，用运

行时检查工具定位，而不是只观察输出是否碰巧正确。（4）对比优化构建和 `-G` 构建的资源报告；该对比说明构建配置影响测量，不能用调试构建作为优化收益基线。

第三章

SIMT、参与集合与内存顺序

本章的学习目标

本章把“线程大致一起运行”改写为可检查的规则：每次协作由谁参与、每个读取由哪个写入产生、什么操作建立先后关系。验收时要能指出错误 mask、缺失屏障、数据竞争和浮点顺序变化，并说明修正后的适用条件。

3.1 SIMT 不提供逐行锁步承诺

SIMT（单指令、多线程）让一组 lane 执行相同指令，每个 lane 仍有自己的数据、谓词和控制流。当 warp 内线程走不同分支时，不同路径需要不同的活跃 lane 集合；短分支还可能被编译为谓词执行。路径代价之和是理解发散的一种近似，不是固定计时公式。若整个 warp 的条件相同，存在分支语句也不等于发生 warp 内发散。

Volta 起的独立线程调度允许更细粒度的调度和重新汇合。因此，在同一个源代码路径中，某一时刻活跃的线程也未必恰好是算法认为的完整逻辑组。依赖相邻线程共享内存写读的旧式 warp 同步代码必须使用有保证的原语。[1]

3.2 Mask 是参与契约，不是自动补零器

`__shfl_sync(mask,v,src)` 在 warp 内交换寄存器值。关键规则是：每个调用线程自己的位必须在 mask 中；mask 指定的所有尚未退出线程必须以同一个 mask 执行匹配的同一种 intrinsic；读取的源 lane 还必须实际参与。否则结果未定义。shuffle 同步寄存器交换，不提供共享或全局内存的屏障语义。[1, 4]

`__activemask()` 是当前指令处活跃 lane 的快照，不会等待其他 lane，也不证明它们是算法规定的参与者。先发散，再用快照构造“本来应该参加的线程集合”，可能把同一逻辑组拆成多个子集。类似地，把一个稀疏 mask 交给 `__shfl_down_sync` 不会将 lane 编号压紧：若 offset 指向不参与的 lane，值仍未定义。`width` 只按连续的二次幂区间分组，不完成稀疏压缩。

先看一个参与集合明确的例子。要求一维 block 的线程数是 32 的倍数，调用前没有线程提前退出；整个 warp 先投票，再由满足谓词的线程广播组内首个 lane 的值：

```
unsigned full = 0xffffffffu;
unsigned lane = threadIdx.x & 31u;
bool valid = i < n;
unsigned members = __ballot_sync(full, valid);
if (valid) {
  float v = x[i];
  int leader = __ffs(members) - 1;
  float first = __shfl_sync(members, v, leader);
  out[i] = first;
}
```

投票由所有 32 个 lane 共同执行，集合因此在发散前定义。`members` 为空时没有线程进入分支，也不会计算非法的源 lane。这个例子展示广播，不能把它机械改为对稀疏成员执行固定 offset 的树归约。部分 warp 的 block、已有发散的调用点、提前 return，均不满足本例前提。

对尾部不足 32 个数据的求和，更简单的办法是启动完整 warp，让越界数据贡献加法单位元零，并让所有 lane 参与：

```
// Called by all lanes of a complete warp, with identical control flow.
__device__ float sum_full_warp(float v) {
  for (int d = 16; d > 0; d >>= 1)
    v += __shfl_down_sync(0xffffffffu, v, d);
  return v;  // only lane 0 is specified as the total
}
```

`shfl_down` 不跨越 warp；超出该 warp 边界的源按原语规则返回本 lane 的值。上述常见树只将 lane 0 的结果作为总和，其他 lane 的返回值不应当作完整归约。第五章把该契约嵌入完整 block 归约。整数、稀疏集合或其他算子可以选择有相应类型/架构支持的 reduce intrinsic、Cooperative Groups 或 CUB，而不必重造未验证的 mask 算法。

3.3 屏障先保证匹配，再讨论数据

`__syncthreads()` 是 block 内屏障；此前的共享和全局内存访问对该 block 内线程具有规定的可见性。它不等待其他 block，也不使多个冲突写入突然变得有定义。`__syncwarp(mask)` 为指定 warp 参与线程提供同步和参与者之间的内存顺序；其 mask 契约仍要满足。

```
// Wrong for a partial block: the predicate differs across threads.
if (threadIdx.x < count) {
  scratch[threadIdx.x] = value;
  __syncthreads();
}

// Define missing data and keep the block-wide rendezvous unconditional.
scratch[threadIdx.x] = threadIdx.x < count ? value : 0.0f;
__syncthreads();
```

即使 `count` 是所有线程看到的同一个常量，`threadIdx.x < count` 仍可能在线程间不同。问题是到达屏障的控制流是否在参与线程间一致，不是某个参数是否相同。条件屏障只有在条件对整个 block 一致、各线程按匹配顺序执行时才安全。教学实现最好把有效性判断放在数据操作上，屏障放在公共路径，且不在屏障前对边界线程提前 return。

循环也要审查匹配：若一个线程执行 3 次屏障，另一个执行 2 次，就没有完整的阶段对应关系。双缓冲不仅要在读取前等待写完，还要在复用缓冲区前等待所有旧读取完成。把 `volatile` 加到共享数组上不会提供这些屏障。

3.4 Barrier、fence 与 atomic 的不同职责

一次非原子写和另一个线程对同一位置的读或写，若没有适用的先后关系，就可能构成数据竞争。即使单次 load/store 看似不可分割，也不能据此推导完整协议正确。

机制	建立的保证	单独不能解决的问题
barrier	一组线程的会合；按原语提供参与者间内存顺序	不同 block 的普通同步；同阶段冲突写
fence	调用线程内内存操作在某个 scope 的顺序约束	让消费者等待；自动发布普通数据；调度进展
atomic	指定位置操作在 scope 内不可分割；可带 order	任意多地址事务；默认完整阶段同步

Scope 回答“哪些线程间适用”：block、device、system；order 回答“其他内存操作如何与本操作排序”：relaxed、acquire、release、acq_rel、seq_cst 等。系统作用域还要求硬件、分配类型和参与设备支持，不能因为地址在 Unified Memory 中就默认 CPU/GPU 原子互操作成立。

传统的 `atomicAdd` 等无后缀 CUDA 原子在设备作用域上提供原子性，默认是 relaxed 顺序；`_block` 和 `_system` 后缀改变适用范围，并受 API/架构支持限制。它们适合独立计数器，不能直接用一个 relaxed flag 发布旁边的普通 payload。`__threadfence_block()`、`__threadfence()`、`__threadfence_system()` 分别对应 block、device、system scope 的顺序栅栏；现代 CUDA 内存模型用 seq_cst 原子 fence 描述其语义。fence 本身不是向所有线程广播一份新数据，也不构成完成通知。[1]

3.5 用发布与获取解释通信

以下片段使用 `<cuda/atomic>`，只说明单生产者、单消费者、flag 初始为零、单次发布且不复用的协议。`flag` 的存储满足 `atomic_ref` 对齐要求，所有并发访问均使用兼容的原子方式，payload 和 flag 都是设备可访问且在整个协议期间存活的内存。

```
__device__ void publish(float* payload, int* flag, float v) {
  cuda::atomic_ref<int, cuda::thread_scope_device> ready(*flag);
  *payload = v;
  ready.store(1, cuda::memory_order_release);
}

__device__ bool try_consume(const float* payload, int* flag,
                            float* result) {
  cuda::atomic_ref<int, cuda::thread_scope_device> ready(*flag);
  if (ready.load(cuda::memory_order_acquire) != 1) return false;
  *result = *payload;
  return true;
}
```

消费者 acquire 若读到该 release 发布的值，就建立相应的同步关系：生产者在 release 前的 payload 写入先于消费者成功 acquire 后的读取。若读到零，函数返回 false，不能读取尚未发布的数据。这个例子不含自旋，也不保证某次消费者一定成功；调度进展与内存顺序是两个问题。多生产者、重复使用 flag、内存释放和跨设备访问都需要额外协议。

适用条件与易错点

不应把本片段拼成“所有 block 等待某个 block”的普通 grid 全局屏障。即使 acquire/release 的内存语义正确，尚未调度的生产者也可能被自旋消费者阻塞。普通分阶段算法优先使用同一 stream 中的 kernel 边界，或显式事件依赖；原子通信只在能证明进展和生命周期时使用。

3.6 浮点正确性与确定性是两件事

浮点加法不满足结合律。在 float 的舍入精度下，表达式

$$(10^{20} + (-10^{20})) + 3.14 \quad 与 \quad 10^{20} + ((-10^{20}) + 3.14)$$

可以分别得到约 3.14 与零。不同 block 对同一个 float 的 `atomicAdd` 顺序随调度变化，因此无数据竞争的原子求和也可能产生不同的末位。固定的分层归约树能改善同一实现的可重复性，但跨 GPU、编译器、融合乘加和数学模式的逐位一致，仍需单独约定。

在舍入到最近、无溢出且忽略异常下溢等条件下，一次运算可写成 $\mathrm{fl}(a+b) = (a+b)(1+\delta)$、$|\delta| \le u$。长度 n 的顺序求和有经典绝对误差模型

$$|\hat{s} - s| \le \gamma_{n-1} \sum_i |x_i|, \qquad \gamma_k = \frac{ku}{1-ku}, \quad ku < 1.$$

平衡树的路径深度约为 $\lceil \log_2 n \rceil$，相应误差界的路径项可更小。若正负项严重抵消，真和 $|s|$ 很小，相对误差仍可能很大。因此测试同时报告最大绝对误差、混合容差通过情况，以及明确分母定义的相对误差；不能只报“误差百分比”。NaN/Inf、非正规数和 fast-math 需要另列策略。

3.7 练习与诊断提示

1. 某 warp 仅 lane 0、2、4、6 有效。解释为何用该稀疏 mask 做 offset 为 1、2、4 的 shuffle 树不等价于四元素求和。提示：mask 不重新编号，第一步的源 lane 可能根本不参与。
2. 给出一个 `count=17`、block 为 32 线程的条件屏障反例。提示：参数是统一值，条件却不是统一真值。
3. 把发布协议的 release/acquire 都改成 relaxed，指出丢失了哪条先后关系。提示：flag 自身的原子性保留，payload 的发布保证消失。
4. 对正负抵消序列比较顺序、树形、double 累加与 float 原子求和，分开记录误差和重复运行差异。提示：不重复的结果未必错误，重复的结果也未必接近高精度参考。

第二部分

解释代价：内存、并行模式与测量

第四章

从地址映射理解内存与矩阵转置

本章的学习目标

本章不要求背诵若干“快内存”名称，而是从 lane 到地址的函数出发，预测全局内存 sector 数与共享内存 bank 映射，再用矩阵转置检验预测。最终应能解释一次布局变更同时影响哪些读写、资源和边界条件。

4.1 存储位置、作用域与生命周期

层次	可编程视角	需要核实的代价
寄存器	每线程私有值，编译器分配	数量限制驻留；长活跃区间增加压力
local memory	每线程私有地址空间	可经缓存访问显存；动态数组和 spill 都可能使用它
shared memory	每 block 共享，显式管理	容量、bank、屏障和缓冲复用
L1 / L2	硬件缓存；L1 接近 SM，L2 由设备共享	命中取决于布局、工作集和竞争
global memory	设备全局地址空间，容量由显存等支撑	合并访问、缓存失效、传输字节
主机内存	CPU 地址空间，可有页锁定或映射	互连带宽、分配成本、驻留与同步

global 是地址空间，HBM 是某些设备使用的物理显存技术；使用 GDDR 的 GPU 也执行 global load。local 名为“局部”，不代表位于快速片上存储。自动数组若不能全部寄存器化，可能分配在 local memory；寄存器 spill 是另一种来源。`ptxas` 的 local 使用与 spill load/store 报告需要分别读，不能将任意 local 数组都称为寄存器溢出。

shared 与缓存可能共享部分片上物理容量，具体分配和可配置上限依架构而变。软件管理 shared 的好处是可明确复用和布局，代价是占用资源并引入同步。若数据只用一次，经过 shared 往往只增加操作。若数据多次使用但本来已命中缓存，共享内存带来的 HBM 节省也可能低于简单估计。[1, 2]

4.2 先列出一个 warp 的地址

设 lane ℓ 访问 float 数组的 $x[b + s\ell]$，字节地址为

$$a_\ell = a_0 + 4(b + s\ell), \qquad 0 \le \ell < 32.$$

对计算能力至少 6.0 的常见标量全局访问情形，可按该 warp 涉及多少个对齐的 32 字节 sector 来做第一步分析。它是这一请求层次的模型，不等于最终 HBM 读写量；缓存、复用和写策略还会改变下层流量。[2]

若起点对齐到 32 字节且 $s = 1$，32 个 float 覆盖 128 字节，涉及 4 个 sector。起点向后偏移一个 float，连续的 128 字节可能跨 5 个 sector；多出的 sector 可能被相邻 warp 重用，因此 DRAM 代价不必按 5/4 增长。若 $s = 32$，各 lane 相隔 128 字节，每个 lane 可落在不同 sector，共请求 32 个 sector。以有用的 128 字节除以请求的 1024 字节，利用率只有 12.5%。这只是所述地址模式的手算，不是实

测带宽。

理解与验证

合并访存是同一 *warp* 在一条内存指令上访问地址的性质，不是“一个线程连续访问了许多元素”。一个线程沿行连续循环，如果相邻 lane 分处不同的行，该时刻仍可能形成跨行步长访问。先写 a_ℓ，再讨论合并。

非对齐还应区分效率和合法性。标量 float 访问有效的 float 地址，未对齐到 sector 通常只是效率问题；把任意偏移指针强制转换成要求 16 字节对齐的 `float4*` 则可能违反访问对齐约束。不能用“缓存会处理”绕过类型对齐和边界要求。

4.3 布局、行间距与子视图

对结构体 `struct P {float x,y,z,m;};` 的数组，只读取 x 时相邻 lane 的地址相隔 16 字节；分离成四个数组（SoA）后，对 x 的访问连续。若每个粒子的所有属性都要一起使用，AoS 的局部性又可能有价值；AoSoA 把小组粒子按属性分块，是二者之间的选择。布局收益依赖消费方式、向量化和是否需要转换，不存在对所有算子最优的排列。

矩阵 $A \in \mathbb{R}^{H\times W}$ 的行优先偏移是 $r\mathrm{ld} + c$。带字节 pitch 的指针应先转为字节地址：

```
const char* base = reinterpret_cast<const char*>(ptr);
const float* row = reinterpret_cast<const float*>(base + r * pitch);
float v = row[c];
```

`pitch` 已是字节数，不能再乘 `sizeof(float)`。子视图还可能使起点从对齐位置偏移，从而破坏向量化假设；检查原分配地址对齐并不够。

4.4 用 bank 公式解释 padding

对现代常见的 32-bank shared memory，以单次 *32* 位标量访问为分析单位，相邻 32 位字映射到相邻 bank：

$$\mathrm{bank}(a) = \left\lfloor \frac{a}{4} \right\rfloor \bmod 32.$$

同一 warp 的一条访问中，多个线程访问同一 bank 的不同字可造成冲突并分批服务；多个线程读取同一个字可使用广播。多个线程无序写同一地址仍是数据竞争，广播例外不赋予它正确性。

令 `float tile[32][32]` 的首地址对应 bank 0。按行读取 `tile[r][lane]` 时，字偏移为 $32r + \ell$，bank 为 ℓ，没有冲突；按列读 `tile[lane][c]` 时，偏移为 $32\ell + c$，所有 lane 落入 bank c，对应不同字，形成经典 32 路冲突。

把行跨度改为 33，列读取的 bank 变为

$$(33\ell + c) \bmod 32 = (\ell + c) \bmod 32,$$

各 lane 分散到 32 个 bank。额外分配量是 $32 \cdot 4 = 128$ 字节。它为这一具体 float 转置布局消除了列冲突，不意味着“任何数组末尾加一都最快”。对 64 位数据、向量指令、Tensor Core 的共享读取和较大请求，硬件会拆成子请求，必须按实际指令和架构重新分析；更复杂布局常用 swizzle，见第八章。

4.5 矩阵转置：读写不能只优化一侧

转置定义为

$$B_{c,r} = A_{r,c}, \qquad A \in \mathbb{R}^{H\times W},\ B \in \mathbb{R}^{W\times H}.$$

朴素实现令相邻 lane 读取同一行的相邻列，输入读取合并；输出地址为 $cH + r$，相邻 lane 相差 H 个 float，写回不合并。若直接交换线程映射，则写回合并但读取跨行。共享 tile 的作用是在线程间重排数据，使全局读和全局写都沿连续方向。

下面的实现把 32×32 tile 交给 32×8 线程，一线程处理四行。为说明边界，所有未对应输入的共享位置显式设零。输入和输出是两个不重叠的连续行优先数组；本实现不支持原地转置。

```
__global__ void transpose32(const float* __restrict__ in,
                            float* __restrict__ out,
                            size_t H, size_t W) {
  __shared__ float tile[32][33];
  size_t x = size_t(blockIdx.x) * 32 + threadIdx.x;
  size_t y = size_t(blockIdx.y) * 32 + threadIdx.y;
  for (unsigned j = 0; j < 32; j += 8) {
    tile[threadIdx.y + j][threadIdx.x] =
        (x < W && y + j < H) ? in[(y + j) * W + x] : 0.0f;
  }
  __syncthreads();
  size_t ox = size_t(blockIdx.y) * 32 + threadIdx.x;
  size_t oy = size_t(blockIdx.x) * 32 + threadIdx.y;
  for (unsigned j = 0; j < 32; j += 8) {
    if (ox < H && oy + j < W)
      out[(oy + j) * H + ox] = tile[threadIdx.x][threadIdx.y + j];
  }
}
// Host contract: block=(32,8), grid=(ceil(W/32),ceil(H/32)); skip H*W==0.
```

证明边界安全时，观察输出的有效条件：$\mathrm{ox} < H$ 和 $\mathrm{oy} + j < W$。被读取的共享位置，对应输入行 $32b_y + t_x$ 和列 $32b_x + t_y + j$，正好就是这两个有效性条件。因而任何有效输出读取的都是本 block 已加载的正确输入。所有线程都先完成加载并到达屏障；不存在边界线程绕开屏障的问题。索引乘法在 `size_t` 中计算，主机仍需检查总大小和 grid 上限。

4.6 如何验证布局收益

建立四个版本：连续复制、朴素转置、未 padding 的共享转置、padding 转置。复制是同规模读写的参考上界之一，不保证达到设备标称带宽。固定同样的 H, W、dtype、时钟条件、计时协议和缓存策略，比较事件时间、global sectors、共享冲突和资源报告。有效带宽按逻辑读写计：

$$\mathrm{BW}_{\mathrm{eff}} = \frac{2HW\,\mathrm{sizeof}(\texttt{float})}{T}.$$

这是有用字节除以时间；profiler 的 DRAM 字节可能因缓存、额外事务而不同，二者需分别命名。

输入包含 1×1、1×1003、33×65、1003×517 和空维度。CPU 参考通过独立的两重循环产生 $B_{c,r}$，不能调用同一个错误索引函数构建预期输出。有限 float 的转置不做算术，复制后通常应逐元素相等；若要保留 NaN payload 等位级信息，需按位检查并明确类型处理。只用方阵且尺寸整齐，会漏掉行列交换和尾 tile 错误。

4.7 主机传输和分配也有生命周期

页锁定（pinned）主机内存可用 `cudaMallocHost` 等分配，使符合条件的异步传输避免 pageable 暂存路径；它占用不可换出的主机资源，应复用缓冲区并计入端到端成本。在传输事件完成之前，CPU 不能覆盖 H2D 输入或读取未完成的 D2H 输出。pinned 本身不建立这些依赖。

Unified Memory 简化地址管理，但页错误、迁移和超量使用会改变访问成本。预取和访问建议需要按工作集与平台支持选择，并通过迁移时间线核实。`cudaMallocAsync`/`cudaFreeAsync` 将分配和释放绑定到 stream 顺序；若另一 stream 使用该地址，必须建立跨 stream 依赖，保证读写都在分配之后且释放之前。它不是对所有队列自动安全的垃圾回收器。

练习与提示。(1) 求 stride 为 1、2、8、32 的 32-lane float 访问所覆盖的 sector 数，分别讨论对齐与偏移一个 float。提示：枚举 $\lfloor a_\ell/32 \rfloor$ 的不同值。(2) 对 `tile[16][17]` 设计 16×16 block，分析一个 warp 的 bank。提示：warp 横跨两行，不能直接套用 32 列例子。(3) 给转置加入输入、输出的独立 ld，在 padding 区填充哨兵并验证其不变。提示：逻辑宽度、输出行数和物理 stride 是三个量。(4) 解释为什么未 padding 的转置可能依旧比朴素版快。提示：全局写合并和共享冲突是同时变化的不同成本。

第五章

Map、归约、Scan 与不规则访问

本章的学习目标

本章把并行程序组织为几种可证明、可组合的模式。除了写出求和，还要能说明单位元、阶段边界和中间数组；除了知道前缀和是 $O(n)$，还要能手算一棵 scan 树并处理跨 block 的偏移。

5.1 Map：独立性比“一元素一线程”更基本

Map 计算 $y_i = f(x_i)$，输出相互独立，通常适合 grid-stride loop。f 可以包含 bias、激活和类型转换；同一线程也可处理多个元素。首先确认读取和写入没有跨线程别名，再考虑对齐、向量化和融合。对 $y_i = x_i + x_{i+1}$，若输出覆盖输入，元素表面独立的写入仍可能破坏别人的读取。

Map 的输入边界检查要发生在 load 之前。先读取越界地址，再用条件选择零，依然非法；应写成条件表达式或显式分支，使无效路径不执行该读取。向量 load 要确保整个向量都有效，尾部改用标量，不能只保护向量的第一个元素。

5.2 归约：把一条依赖链变为树

归约将 n 个值组合成一个值。若操作 $\circ$ 有结合律和单位元 e，可以重组为树：加法用零，乘法用一，最大值可用负无穷。NaN 处理、溢出和浮点近似会改变这一数学前提，应作为接口的一部分。

顺序求和做 $n-1$ 次加法，关键依赖路径同样长；平衡树仍做 $O(n)$ 工作，但关键路径缩短为 $O(\log n)$。GPU 上通常按四层组织：每线程局部累加、warp 内寄存器交换、block 内 warp 结果汇总、跨 block 的下一阶段。它用更多并行状态和中间结果缩短依赖链，而不是减少数学加法次数。

以下教学实现要求一维 block，实际线程数与模板参数相同，B 在 32 到 1024 之间且为 32 的倍数；设备每块线程上限仍需查询。输入不与输出重叠，所有线程都执行全部 collective，不得在到达 warp 归约前提前 return。

```
template<int B>
__global__ void reduce_partials(const float* __restrict__ x,
                                float* __restrict__ partial,
                                size_t n) {
  static_assert(B >= 32 && B <= 1024 && B % 32 == 0,
                "B must contain complete warps");
  __shared__ float warp_total[B / 32];
  unsigned t = threadIdx.x;
  float v = 0.0f;
  size_t step = size_t(gridDim.x) * B;
  for (size_t i = size_t(blockIdx.x) * B + t; i < n; i += step)
    v += x[i];
  for (int d = 16; d > 0; d >>= 1)
    v += __shfl_down_sync(0xffffffffu, v, d);
  if ((t & 31u) == 0) warp_total[t / 32] = v;
  __syncthreads();
  if (t < 32) {
    float w = t < B / 32 ? warp_total[t] : 0.0f;
    for (int d = 16; d > 0; d >>= 1)
      w += __shfl_down_sync(0xffffffffu, w, d);
    if (t == 0) partial[blockIdx.x] = w;
  }
```

```
}
```

尾部没有数据的线程仍带着零参加归约，因此固定 full mask 的源 lane 始终存在。第一次 lane 0 写出各 warp 的和；block 屏障使第一 warp 随后读取这些值有定义。最终仅由线程零写出本 block 的完整总和。

跨 block 的结果用下一次 kernel 继续归约；同一 stream 保证阶段依赖，不需主机在两次提交之间等待。主机可以递归缩小 partial 数组直到一个结果。$n = 0$ 的和应显式定义为零，若结果留在设备，可用适当的同 stream 初始化实现；不能返回未初始化标量。$n = 1$ 也需满足同样的输出契约。

5.3 归约的代价与准确性选择

第一阶段读 $4n$ 字节并写 $4G$ 字节 partial，下一阶段继续读写 partial；这是逻辑字节模型，不含 cache 和事务放大。增加每线程元素数可以减少 partial 和 launch，却让单线程局部依赖更长。使用多个独立累加器可提高 ILP，但增加寄存器并改变加法顺序。

采用 float、double 或补偿求和是不同数值/性能方案。不能先用 double 基线、后用低精度优化，再把全部差异称为并行结构的加速。固定树可使同一配置重复更稳定，改变 grid 仍会改变分组与结果。生产实现应至少比较 CUB 的 `DeviceReduce`，并记录其版本、累加类型与确定性契约；库的成熟度不能代替本应用的输入和误差测试。[5]

5.4 Scan：每个元素都需要前面的信息

对序列 $x_0, \dots, x_{n-1}$，包含式 scan 输出

$$y_i = x_0 + \cdots + x_i,$$

排除式 scan 则输出 $z_0 = 0$、$z_i = x_0 + \cdots + x_{i-1}$。若 x_i 是是否保留元素的 0/1 标志，z_i 就是被保留元素应写入的位置，最终总和给出输出长度。这解释了 scan 在压缩、稀疏索引、radix sort 和路由中的作用。

以 $[3, 1, 7, 0, 4, 1, 6, 3]$ 为例，排除式结果为

$$[0, 3, 4, 11, 11, 15, 16, 22],$$

总和为 25。Blelloch scan 先 up-sweep 汇总树，再 down-sweep 把每个节点之前的前缀传向子节点。[6]

阶段	长度为 8 的数组内容
初始	3, 1, 7, 0, 4, 1, 6, 3
up-sweep，间距 1	3, 4, 7, 7, 4, 5, 6, 9
up-sweep，间距 2	3, 4, 7, 11, 4, 5, 6, 14
up-sweep，间距 4	3, 4, 7, 11, 4, 5, 6, 25
根设零后 down-sweep	0, 3, 4, 11, 11, 15, 16, 22

down-sweep 的一个内部节点中，左孩子暂存左子树总和 L，父节点携带此前前缀 P：把左孩子设成 P，右孩子设成 $P + L$。这一步不是交换后继续沿用旧父值的普通归约，而是在传播前缀。树的每层必须等待上一层所有共享写入完成。

5.5 一个可追踪的 block scan

下例选用 N 个线程处理 N 元素，N 是二次幂且不超过设备每块线程上限。非整齐尾 tile 用零填充，保存根总和后再设零。它追求步骤清楚，未做 bank 优化；高性能代码可以用 warp scan 和分层结构替换。

```
template<unsigned N>
__global__ void scan_tiles(const float* in, float* out,
                           float* totals, size_t n) {
  static_assert(N >= 2 && N <= 1024 && (N & (N - 1)) == 0,
                "N must be a supported power of two");
  __shared__ float s[N];
  unsigned t = threadIdx.x;
  size_t base = size_t(blockIdx.x) * N;
  s[t] = base + t < n ? in[base + t] : 0.0f;
  __syncthreads();
  for (unsigned d = 1; d < N; d <<= 1) {
    unsigned k = (t + 1) * (2 * d) - 1;
    if (k < N) s[k] += s[k - d];
    __syncthreads();
  }
  if (t == 0) { totals[blockIdx.x] = s[N - 1]; s[N - 1] = 0.0f; }
  __syncthreads();
  for (unsigned d = N / 2; d > 0; d >>= 1) {
    unsigned k = (t + 1) * (2 * d) - 1;
    if (k < N) {
      float left = s[k - d];
      s[k - d] = s[k];
      s[k] += left;
    }
    __syncthreads();
  }
  if (base + t < n) out[base + t] = s[t];
}
```

每一层参与加法的线程不同，但屏障在条件之外，所有 block 线程执行相同的层数。按 $1/2 + 1/4 + \cdots$ 求和，up-sweep 总共 $N-1$ 次操作，down-sweep 同量级，工作量为 $O(N)$；层数为 $O(\log N)$。Hillis–Steele 的直接逐层前缀法更直观，但若每层所有位置都更新，会做 $O(N \log N)$ 工作，且同层读旧值/写新值需要双缓冲或恰当同步。两者不应只因层数相同就视为等价。

跨 block 采用三阶段方案：扫描每 tile 并保存 totals；对 totals 递归做排除式 scan 得到 offsets；把 offsets[b] 加到 tile b 的每个有效输出。最后一个阶段多读写一次长度为 n 的输出，换取容易证明的全局依赖。若把 totals 做成包含式 scan，就会错误地把本 tile 总和也加入前缀。

更先进的单遍 scan 用 look-back 等机制减少数据往返，但其状态发布、观察顺序和前进保证不是一个普通自旋循环可以替代的。Merrill 与 Garland 的 decoupled look-back 是理解该方向的原始资料；实际应用优先比较 CUB。[7, 5] 浮点加法只近似满足结合律，各种 scan 的舍入分组会不同；离散计数可用足够宽的整数，但也需防总计数溢出。

5.6 直方图与热点分布

直方图把输入映射到桶 $h_{b(x_i)}$ 并加一。直接 global atomic 的问题取决于分布：输入集中在一个桶时高度争用，均匀分布时争用小得多。每 block 使用共享直方图时，流程必须是“初始化桶、block 屏障、原子更新、block 屏障、汇总到全局”；漏掉初始化后的屏障，更新可能被尚未完成的清零覆盖。

每 warp 私有桶可降低跨 warp 争用，但共享内存用量为桶数乘 warp 数再乘计数大小，桶多时会限制驻留。排序后分段计数可规整热点，却增加排序流量。应对均匀、单热点和长尾分布分别测试，报告桶数、计数位宽和端到端成本，不能仅以一种随机均匀输入决定算法。

5.7 邻域计算与 Scatter/Gather

二维五点 stencil 为

$$y_{i,j} = ax_{i,j} + b(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1}).$$

一个 $T_x \times T_y$ tile 若加载一圈 halo，共需 $(T_x + 2)(T_y + 2)$ 个值，冗余比例是该数除以 T_xT_y。tile 太小则 halo 比例高；太大则共享内存和线程资源昂贵。零填充、周期和镜像是不同边界条件，不能为提高性能悄悄改写算子。原地 stencil 通常也需要额外缓冲，以免覆盖邻居读取。

Gather 按索引读，一般允许多个线程读取同一输入，但要检查索引范围和局部性。Scatter 按索引写；多个线程指向同一地址时，若语义是求和需原子或分段归约，若语义是覆盖则必须定义胜出规则。对 embedding 与 MoE 路由，可先按目标重排使随机 scatter 变为分段操作，但重排、逆映射和额外存储都应计入完整时间。

5.8 练习与验收

1. 对 $n = 0, 1, 31, 33, 257, 1003$ 验证分层归约，使用独立 double 参考，并改变 grid。提示：逻辑尾部补零，物理 warp 仍须完整；空结果要初始化。
2. 手算两个 tile 对 $[3, 1, 7, 0, 4, 1, 6, 3]$ 的四元素排除式 scan。提示：tile totals 为 11、14，offsets 为 0、11。
3. 用 0/1 flags 实现稳定压缩，测试全零、全一及空输入。提示：最终长度为最后一个排除式前缀加最后一个 flag，空输入需单独处理。
4. 为直方图设计同样大小但不同争用度的三种输入；先预测最可能的瓶颈，再测量。提示：输入大小相同不等于原子冲突相同。

第六章

Roofline、延迟隐藏与可复现测量

本章的学习目标

本章把“快”改写成可检验的命题 :先算工作量和数据流, 再用统一的计时协议、统计量和 profiler 证据解释结果。读完后，应能区分 kernel 时间与端到端时间，判断算术、带宽、延迟或提交开销哪一个更可能限制性能。

6.1 先算算术强度，再猜瓶颈

算术强度是从慢速内存搬运一个字节平均完成多少浮点操作：

$$AI = \frac{\text{FLOPs}}{\text{DRAM bytes}}, \qquad P \le \min\left(P_{\text{peak}}, \text{BW}_{\text{DRAM}} AI\right). \tag{6.1}$$

这是上界模型，不是测量结果。矩阵乘法 $C = AB$（方阵边长 N，FP32）有 $2N^3$ 次浮点操作；朴素实现至少读写约 $12N^2$ 个字节，故其逻辑算术强度约为 $N/6$。分块让 A 和 B 被多个线程复用，提高强度；它同时会增加共享内存、寄存器和边缘 tile 的代价。

实际字节数必须说明口径：是算法的逻辑读写，还是 profiler 统计的 sector/transaction？cache 命中、写分配和对齐都会改变二者的关系。若实测点接近带宽斜线，优先检查地址合并、重复加载和布局；若接近计算上界，才进一步检查指令吞吐、Tensor Core 利用率和依赖链。Roofline 不能证明某个优化有效，只有在同一 shape、dtype 和计时范围下的对照才可以。

6.2 延迟隐藏不是“把 occupancy 调到最高”

GPU 用线程级并行（TLP）在一个 warp 等待内存时调度另一个 warp；每个线程保持多个独立累加器或展开循环，则增加指令级并行（ILP）。粗略地说，可运行 warp 数与每个 warp 的独立工作共同决定是否能填满 load、math 或 tensor 管线。寄存器过多、共享内存过大或 block 过大都会减少驻留机会；过度限制寄存器又可能引入 local memory spill。

理解与验证

Occupancy 是“隐藏延迟的机会”，不是速度本身。一个低 occupancy 的 GEMM 可能因寄存器复用和 Tensor Core 利用率较高而更快；一个高 occupancy 的 kernel 若所有 warp 都等同一条依赖链，仍可能停滞。把 occupancy、stall reason、内存吞吐和实际时间放在同一张记录中。

6.3 计时协议：先明确时间边界

CUDA Events 记录的是同一 stream 中两个事件之间的设备时间。一个最小的 kernel 计时流程如下；事件停止后必须同步，不能用主机提交函数返回时间代替设备执行时间。

```
for (int i = 0; i < 5; ++i) kernel<<<grid, block, 0, stream>>>(args...);
CUDA_CHECK(cudaStreamSynchronize(stream));
cudaEvent_t start, stop;
CUDA_CHECK(cudaEventCreate(&start));
CUDA_CHECK(cudaEventCreate(&stop));
```

```
for (int r = 0; r < 31; ++r) {
  CUDA_CHECK(cudaEventRecord(start, stream));
  kernel<<<grid, block, 0, stream>>>(args...);
  CUDA_CHECK(cudaEventRecord(stop, stream));
  CUDA_CHECK(cudaEventSynchronize(stop));
  float ms = 0.0f;
  CUDA_CHECK(cudaEventElapsedTime(&ms, start, stop));
  samples.push_back(ms);
}
```

端到端时间则应从 host steady clock 或一对覆盖 H2D、kernel、D2H 的事件定义，并明确是否包含分配、布局转换、CPU 参考和最终同步。两种时间不能混在一列再宣称“加速”。预热用于排除首次 JIT、缓存和 lazy initialization；正式报告仍应保留原始样本，给出 median、p95 及异常值规则。

6.4 正确性、统计和单位一起记录

对于每个版本，至少保存 e_{abs}、e_{rel}、NaN/Inf 数量、随机种子、shape、stride、dtype、warm-up/repeat、grid/block、静态 shared 和寄存器。吞吐的换算为

$$\text{GB/s} = \frac{\text{读字节} + \text{写字节}}{10^9 T_{\text{s}}}, \qquad \text{GFLOP/s} = \frac{\text{FLOPs}}{10^9 T_{\text{s}}}.$$

浮点归约的误差预算应随求和深度和 $\sum |x_i|$ 变化；把一个低精度结果与 double 参考的差异直接当成线程结构误差是不成立的。固定 seed 只保证输入可重建，不能保证不同架构或不同并行分组的浮点顺序相同。

字段	用途
GPU、compute capability、driver、CUDA	解释指令选择、资源上限和 JIT 行为。
shape、stride、dtype、layout	确认比较对象与访问模式相同。
warm-up、repeat、统计量	区分首次开销、抖动与稳态时间。
baseline、commit、编译选项	防止比较对象和代码版本漂移。
误差、NaN/Inf、seed	证明数值契约与结果可重建。

6.5 从时间线到单 kernel 指标

Nsight Systems 回答“什么时候在做什么”：CPU/GPU 时间线、stream overlap、launch 间隙、通信与计算是否重叠。Nsight Compute 回答“这个 kernel 为什么慢”：内存吞吐、cache 命中、warp stall、寄存器/shared、Tensor Core 管线和 Roofline 位置。`ptxas -v` 给出编译期资源；`cuobjdump` 或 `nvdisasm` 才能确认预期的 `mma`、`ldmatrix` 或异步搬运指令是否真的生成。工具版本和指标名会变化，应把命令、版本和原始报告一同保存。[8, 9, 4]

适用条件与易错点

分析器会改变缓存、replay 和提交开销。不要把 Systems/Compute 运行中的 harness 时间当作正常生产时间；先在无分析器条件下保存正确性和计时基线，再用 profiler 回答具体问题。

6.6 第六章检查清单与练习

- 能从伪代码列出 FLOPs、逻辑字节数和算术强度，并说明口径。
- 能区分 kernel 计时、端到端计时、median 与 p95。
- 能把 Systems 的时间线观察映射到 Compute 的一个可证伪假设。

· 能说明 occupancy、TLP、ILP 与最终吞吐不是同义词。

练习：为 SAXPY、两阶段归约和 GEMM 各写一条 JSON 记录；固定输入和 CUDA stream，先记录 5 次预热、31 次样本，再分别计算 median、nearest-rank p95、GB/s、GFLOP/s 和误差。把一个“更快”的结果改写成包含 shape、dtype、版本和计时边界的可复核结论。

第七章

优化方法：把收益假设变成实验

本章的学习目标

优化不是技巧名词的清单，而是一条可回归的因果链：先锁定正确性和基线，再用数据流、资源和 profiler 解释瓶颈，每次只改变一个主要因素，最后检查所有输入与架构。读完后，应能写出一个“若……则……”的性能假设，并说明它可能在哪些形状上退化。

7.1 按证据排序的优化流程

推荐顺序如下：

1. 用 CPU 或高精度参考覆盖零长度、非整齐长度、极值和 NaN/Inf 契约；对设备代码先做 Compute Sanitizer 检查。
2. 固定输入、dtype、shape、编译选项和计时边界，建立有 warm-up 的 baseline，并同时保存 kernel 与端到端时间。
3. 用 Roofline 和 Nsight 判断第一限制是带宽、计算、延迟、同步、launch 还是通信，不因一个 occupancy 数字改变方向。
4. 优先改数据移动：合并访存、布局、复用、tile、融合；随后再调 block、warp primitive 和 ILP。
5. 目标架构支持时才加入 Tensor Core、低精度、`cp.async`/TMA 或持久化路径，并保留可移植 fallback。
6. 对全部回归矩阵重复正确性和性能测试，记录失败 shape、资源变化和适用边界。

7.2 常见方法及其收益假设

方法	收益假设、代价与验证
向量化（`float4`/`half2`）	减少地址指令并改善事务利用；需要对齐、合法长度和兼容布局，尾部必须回退标量。
shared/register tiling	用片上复用减少 DRAM 往返；会增加 shared、寄存器、同步和 bank conflict 风险。
warp shuffle/投票	在 warp 内交换数据而不写 shared；mask 必须与实际参与集合一致。
融合	省去中间张量写回和 launch；寄存器压力、不同 tile 形状和多消费者会使收益消失。
异步流水/双缓冲	把搬运与计算重叠；必须证明 wait、barrier、stage 生命周期和尾部状态。
Tensor Core/低精度	提高矩阵吞吐或降低字节数；需要匹配 shape、layout、累加类型和 scale，误差需重新验收。

7.3 编译器提示不是免费的性能开关

`__restrict__` 是别名承诺，写错会导致未定义行为；`constexpr`、模板参数和 `#pragma unroll` 只有在循环边界和资源预算允许时才有益。`--use_fast_math` 可能改变近似函数、舍入和次正规数处理，应把它视为改变数值算法的选项。`-maxrregcount` 或过紧的 `__launch_bounds_` 可能将寄存器挤到 local memory；用 `-Xptxas=-v` 与 profiler 检查 spill，而不是只看编译成功。

7.4 融合与重排：少一次往返仍需付出代价

对 $z_i = \mathrm{cast}(\mathrm{gelu}(x_i + b_i))$，三个独立 kernel 要多次写读中间数组；一个融合 kernel 可以让每个元素在寄存器中完成全部操作。收益成立的前提是中间值没有其他消费者，且融合后的寄存器、shared 和指令量没有阻止足够的并发。若多个后续算子复用同一中间结果，物化一次可能更便宜；若两个算子需要完全不同的 tile 形状，强行融合也可能破坏合并访存。

评价融合时同时记录总字节数、launch 数、寄存器/shared、occupancy、kernel/端到端时间和误差。不能只用某一个长向量的 kernel 时间概括所有形状；小输入可能由 launch 主导，大输入可能由带宽或算术主导。

7.5 第七章检查清单与练习

- 每次优化都能写出收益假设、资源代价和可能退化的输入范围。
- 能区分向量化、tile、融合和异步流水各自改变的数据流。
- 能解释 `--use_fast_math`、寄存器上限和 fast path 的正确性风险。
- 能为融合 kernel 设计 shape、dtype、layout 和误差回归矩阵。

练习：从 bias + GELU 的两个 kernel 开始实现融合版本；固定输入后比较逻辑 DRAM 字节、寄存器、端到端时间和误差，并用一个短向量和一个大矩阵寻找“融合没有收益”的边界。

第三部分

构造算子：矩阵、归一化与运行时

第八章

GEMM：从数据复用到 Tensor Core

矩阵乘法是检验前七章知识能否连起来的例子。索引决定结果是否正确，布局决定一条加载服务多少个线程，tile 决定数据被复用几次，同步决定共享缓冲能否重用，数值格式决定计算的含义。本章先构造可解释的 FP32 实现，再说明生产库为何还需要 warp tile、Tensor Core 和架构特化。教学实现用于建立因果关系，比较对象仍应包括同条件下的 cuBLASLt 或 CUTLASS，而不是只比较自己的两个版本。[1, 2, 10, 11]

8.1 先把矩阵乘法的契约写完整

一般矩阵乘法（general matrix multiplication，GEMM）可写为

$$C_{m,n} \leftarrow \alpha \sum_{k=0}^{K-1} A_{m,k} B_{k,n} + \beta C_{m,n}. \tag{8.1}$$

本章代码先取 $\alpha = 1, \beta = 0$，输入不重叠，输出不与输入别名，矩阵均为行优先。若存在行尾填充，其元素地址分别为

$$A[m\,\mathrm{lda} + k], \qquad B[k\,\mathrm{ldb} + n], \qquad C[m\,\mathrm{ldc} + n],$$

并要求 $\mathrm{lda} \geq K, \mathrm{ldb} \geq N, \mathrm{ldc} \geq N$。leading dimension 的单位是元素，不能与字节 pitch 混用。下列实现进一步取连续布局 $\mathrm{lda} = K, \mathrm{ldb} = N, \mathrm{ldc} = N$，但地址乘法先提升到 `size_t`。主机必须验证维度非负、容量乘法不溢出，并在 $M = 0$ 或 $N = 0$ 时直接返回。

当 $K = 0$ 时空和为零，因此本章契约要求非空输出全为零；不能因没有输入元素就跳过写回。一般 $\beta \neq 0$ 的实现则要保留缩放后的旧 C。这一差别也说明为什么 benchmark 必须注明是否读入旧输出。

```
__global__ void gemm_naive(const float* __restrict__ A,
                           const float* __restrict__ B,
                           float* __restrict__ C,
                           int M, int N, int K) {
  size_t row = size_t(blockIdx.y) * blockDim.y + threadIdx.y;
  size_t col = size_t(blockIdx.x) * blockDim.x + threadIdx.x;
  if (row >= size_t(M) || col >= size_t(N)) return;
  float acc = 0.0f;
  for (int k = 0; k < K; ++k)
    acc = fmaf(A[row * K + k], B[size_t(k) * N + col], acc);
  C[row * N + col] = acc;
}
```

朴素版本每个输出做 K 次乘加，按一次乘、一次加各算一个 FLOP，常用工作量计为 $2MNK$。它在源码层面发出约 $2MNK$ 次输入加载；缓存和 warp 内广播可能减少实际 DRAM 流量，因此不能把这直接当作硬件读字节数。理想情况下，每个矩阵只穿过 DRAM 一次，FP32 且 $\beta = 0$ 的数据量下界为

$$B_{\min} = 4(MK + KN + MN), \qquad AI_{\text{ideal}} = \frac{2MNK}{B_{\min}}. \tag{8.2}$$

这个下界描述整个问题的充分复用，不是朴素程序必然达到的字节数。对方阵它给出 $N/6$ FLOP/byte；如果把它贴到任何 GEMM 的 Roofline 点上，就会掩盖不同实现的实际流量差别。

8.2 共享内存分块：同步也是算法的一部分

用一个 $T \times T$ 的 CTA 负责输出的一个 tile，每轮共同加载 A 与 B 的两个输入 tile。这样每个共享元素可供约 T 个输出使用。为让边界也满足屏障契约，所有线程执行相同的轮数，越界输入填零，最后只对有效输出写回：

```
template<int T>
__global__ void gemm_tiled(const float* __restrict__ A,
                           const float* __restrict__ B,
                           float* __restrict__ C,
                           int M, int N, int K) {
  static_assert(T > 0 && T * T <= 1024);
  __shared__ float As[T][T], Bs[T][T];
  int tx = threadIdx.x, ty = threadIdx.y;
  size_t row = size_t(blockIdx.y) * T + ty;
  size_t col = size_t(blockIdx.x) * T + tx;
  float acc = 0.0f;
  // Required launch: block == dim3(T,T,1).
  for (size_t base = 0; base < size_t(K); base += T) {
    size_t ak = base + tx, bk = base + ty;
    As[ty][tx] = (row < size_t(M) && ak < size_t(K))
                 ? A[row * K + ak] : 0.0f;
    Bs[ty][tx] = (bk < size_t(K) && col < size_t(N))
                 ? B[bk * N + col] : 0.0f;
    __syncthreads();
    #pragma unroll
    for (int k = 0; k < T; ++k)
      acc = fmaf(As[ty][k], Bs[k][tx], acc);
    __syncthreads();
  }
  if (row < size_t(M) && col < size_t(N))
    C[row * N + col] = acc;
}
```

第一个屏障使所有生产者完成共享写入，消费者才能读 tile；第二个屏障确保所有读者已结束，下一轮生产者才能覆盖同一存储。输出越界的线程也可能负责加载其他线程要读的输入，因而不能在内层循环前提前返回。$K = 0$ 时循环不执行，累加器的零仍会被写回。$T = 16$ 是便于教学的候选值，不是性能承诺；主机还须查询设备线程数、共享内存和 grid 各维上限，不能只凭静态断言判断 launch 合法。

将一个输出 tile 的每轮计算与输入搬运相除，忽略边缘、写回与缓存，有

$$AI_{\text{tile,input}} = \frac{2T^3}{4(2T^2)} = \frac{T}{4}\ \text{FLOP/byte}.$$

这是 CTA 内一轮的输入复用模型，不能与式 (8.2) 的全矩阵下界混为一谈。增大 tile 可能减少重复加载，也可能让 CTA 数量太少、边缘浪费增加，或让资源限制阻止足够的 block 驻留。

手算边界。取 $M = 3, N = 5, K = 7, T = 4$。输出需要两个 CTA，内层执行两轮；第二轮的 $k = 7$ 加载填零。第二个 CTA 中只有一列输出有效，但全部 16 个线程仍参加两次屏障。若在 $\text{col} \geq 5$ 时提前返回，剩余线程的共享输入与同步假设都会被破坏。这个小例子应先用整数可精确表示的数核对，再测随机浮点输入。

8.3 Register tiling：把共享读取换成线程内复用

共享分块只解决了 global-to-shared 的复用。让一个线程维护 $T_M \times T_N$ 个输出累加器，则每个 k 只需加载 T_M 个 A 值和 T_N 个 B 值，做一个外积更新：

$$r_{i,j} \leftarrow \text{fma}(a_i, b_j, r_{i,j}), \quad 0 \leq i < T_M, \quad 0 \leq j < T_N. \tag{8.3}$$

$T_M = T_N = 4$ 时，8 次共享读取支持 16 次乘加；若独立计算这 16 个输出而不在寄存器重用，每步要读取 32 个操作数。该比值只计源码层面的读取次数，实际带宽还受广播、指令合并和 bank 布局

影响。

CTA tile 用 (B_M, B_N, B_K) 表示。忽略装载与计算分工细节，一个资源预算是

$$S_{\text{CTA}} = p\, b(B_M B_K + B_K B_N) + S_{\text{pad}} + S_{\text{barrier}}, \tag{8.4}$$

$$R_{\text{thread}} \gtrsim T_M T_N + T_M + T_N + R_{\text{index}}, \tag{8.5}$$

其中 b 为每个输入元素的字节数，p 是缓冲 stage 数，R 以 32 位寄存器单位粗估。FP32、$B_M = B_N = 64, B_K = 16$ 的单缓冲输入需 8192 字节共享内存；若 256 个线程各算 4×4 个输出，仅累加器就有 4096 个 32 位数。双缓冲把输入共享存储翻倍，却不自动减少累加器。编译器的额外地址变量、展开、分配粒度和 spill 只能由资源报告确认，这两个式子不能取代 `ptxas` 的结果。

线程到寄存器 tile 的映射还要使一组 lane 的 global load 连续，并使共享读取的 bank 分布合适。把数学上邻近的 4×4 tile 原样分给线程，未必产生理想访问。生产 GEMM 因而采用 CTA tile、warp tile、线程或指令 tile 的层级布局，配合向量装载、padding 或 swizzle；第 四 章的地址公式比“tile 越大越好”更能指导选择。

理解与验证

可证伪的假设是：“register tiling 减少了共享装载，因此在相同输入和精度下，应降低每次有效乘加的共享指令或共享流量，且没有明显 spill。”如果时间变差，检查寄存器、活跃 CTA、共享 bank 指标与指令依赖，再决定是否缩小线程 tile。低 occupancy 本身不能证明这个实现更慢。

8.4 WMMA：一个 warp 共同拥有一个矩阵片段

Tensor Core 执行固定形状和类型的矩阵乘加。CUDA 的 WMMA 接口用 fragment 表示分布在 warp 各 lane 寄存器中的矩阵片段；fragment 的内部元素到 lane 映射不应被应用程序猜测。下面是一个完整 $16\times16\times16$ 乘法步骤的接口片段，不是带边界的完整 *kernel*：

```
#include <mma.h>
#include <cuda_fp16.h>
using namespace nvcuda;
wmma::fragment<wmma::matrix_a, 16,16,16,
               __half, wmma::row_major> a;
wmma::fragment<wmma::matrix_b, 16,16,16,
               __half, wmma::row_major> b;
wmma::fragment<wmma::accumulator, 16,16,16, float> c;
wmma::fill_fragment(c, 0.0f);
wmma::load_matrix_sync(a, a_tile, lda);
wmma::load_matrix_sync(b, b_tile, ldb);
wmma::mma_sync(c, a, b, c);
wmma::store_matrix_sync(c_tile, c, ldc, wmma::mem_row_major);
```

其前提必须同时成立：目标 GPU 与编译目标支持该形状和类型；完整 warp 的所有 lane 都执行各个 collective，并对指针、stride、布局和模板参数达成一致；三个 tile 的存储范围完整可访问。依据 CUDA Programming Guide 对此 WMMA 接口的约束，加载/存储指针须满足 256 位对齐，半精度输入的 leading dimension 为 8 的倍数，FP32 输出为 4 的倍数；这些要求不能套用为所有 PTX MMA 或所有库的统一规则。[1]

对于 M, N, K 非指令 tile 倍数的情况，不能用 `if (row<M)` 让部分 lane 跳过 `mma_sync`。一种易审计的教学方案是：CTA 把边缘输入带谓词地搬入满足布局与对齐要求的共享 tile，越界元素填零；完整 warp 做 WMMA；先把输出存到完整临时 tile，再由线程带谓词地写回有效元素。若存在 CTA 协作装载，还需相应的 CTA 屏障；`load_matrix_sync` 的名字不表示它替整个 CTA 等待了生产者。这个安全方案增加了边缘开销，生产库通常区分完整 tile 与尾部路径。

fragment 也不是稳定的跨架构 ABI：不同架构的内部表示可能不同，不能随意在采用不同目标编译的对象文件之间传递 fragment；先存为普通矩阵再跨接口传递更稳妥。只有对所有 fragment 元

素做同一个逐元素变换时，才可按文档允许的方式遍历其存储；需要按逻辑 (m, n) 加 bias 时，应使用有明确坐标映射的 epilogue。[1, 11]

8.5 精度格式改变了参考问题

FP16 或 BF16 输入配 FP32 累加是常见的混合精度路径，但这不意味着结果等于 FP32 输入 GEMM。输入转换已经造成误差，累加顺序和输出转换也会改变结果。对一个长度 K 的点积，朴素舍入模型的绝对误差与 $\sum_k |a_k b_k|$ 相关；强抵消时，即便绝对误差小，相对误差仍可很大。因此验证必须同时使用 atol + rtol$|y_{\text{ref}}|$ 的逐元素判据、最大绝对误差，并覆盖抵消、大小量混合与接近零输出。

TF32 是 Tensor Core 运算格式，不是普通应用数组的 32 位浮点存储类型。它保留 FP32 的指数范围，使用较少的有效尾数位；Ampere 文档给出 8 位指数与 10 位显式尾数位。显式 WMMA TF32 路径有自己的形状和转换要求，例如相应接口要求先按文档用 `__float_to_tf32` 转换输入；不能把上面的 half fragment 只改名字就当成 TF32 实现。库路径则通过 math mode、compute type 与算法选择控制是否允许 TF32。比较“FP32 GEMM”时应记录这些设置，而不猜测框架版本的默认值。[12, 1, 10]

FP8 进一步需要 scale 和格式约定。E4M3 与 E5M2 的指数、尾数和特殊值编码不同；E4M3 的常见有限数变体不提供无穷大，而 E5M2 可编码无穷大，其他 FNUZ 等变体又有不同约定。字节宽度相同不能保证二进制兼容。论文格式、CUDA 类型、库 descriptor 和硬件指令需要逐一对应；实际量化误差与 scale 粒度见第 九 章。[13]

8.6 继续优化之前，先写收益与代价

Split-K。当 M, N 很小、K 很大时，输出 tile 数量不足，可把 K 分成 P 份并行得到 $C^{(p)}$，再求和。新增约 PMN 个临时值的写回与读取、归约 launch 或原子竞争；βC、bias 与非线性激活必须在最终归约后正确应用一次。尤其不能对每个 partial output 先做 ReLU 再求和。原子 Split-K 还可能改变可复现性。

Epilogue fusion。在累加器写回时做 bias、激活或 cast，可省去中间矩阵的一次写回后重读和一个 launch。收益假设是中间值无其他必要消费者，且 epilogue 的寄存器与指令代价没有破坏主循环；对小矩阵可能主要节省 launch，对大矩阵则要查看内存与 Tensor Core 利用率。[11, 10]

多 stage 流水。输入 tile $t+1$ 的搬运可与 tile t 的计算重叠，但 stage 会增加式 (8.4) 的存储和同步状态。若计算已足够隐藏延迟，继续加 stage 可能无益；若 K 太短，流水填充和排空可能主导。异步操作的完成语义见第 十二 章。

持久化调度。让 CTA 循环领取多个 tile，可控制工作分配、重用与调度开销；它并不意味着把数据永久留在寄存器中。小 tile 数、不均匀边缘或与其他 stream 的资源竞争都可能使它退化。只有时间线、资源计数与多 shape 测量同时支持，才能把它作为默认路径。

8.7 渐进实验与解题提示

使用本项目 `examples/` 的统一输入和计时协议，先对朴素与共享分块版本比较正确性，再进入 register tiling 与库基线。应测试 $(M, N, K) = (3, 5, 7)$、$(127, 129, 65)$、一个规则方阵、一个很小 M, N 而很长 K 的形状，以及 $K = 0$。同一比较内固定输入、dtype、输出契约、warm-up、重复次数和计时区间。资源报告、sanitizer 与事件时间分别保存；未在 GPU 执行的项标为待验证。

1. 为 $B_M = 128, B_N = 64, B_K = 32$ 的 FP16 tile 计算单、双、三 stage 的输入共享字节数。提示：先

用式 (8.4) 忽略 padding，再解释为什么该数小于容量仍不能保证好性能。

2. 在共享分块代码中删除最后一个屏障，构造下一轮写入可能覆盖上一轮读的时序。提示：不要以一次运行正确作为反证；依赖必须对所有合法调度成立。
3. 给 $\beta \neq 0$ 的 Split-K 写出数学上正确的归约顺序，并估算临时数组流量。提示：区分线性缩放、只应用一次的旧 C 和不能分配到部分和上的非线性。
4. 用相同数据分别允许和禁用库的 TF32 路径。提示：除时间外，记录允许的计算类型、算法、版本和误差；不把两者自动归为同一精度基线。

完成本章的标志，是能为每个 tile 画出生产、消费、复用与写回的顺序，并说明一个加速假设在什么形状或资源约束下会失败。

第九章

训练与推理算子：稳定性、融合与在线计算

GEMM 的复用以乘加为中心，Softmax 与归一化则以归约为中心。一个算子即使只有几行数学式，也可能在中间张量、同步、特殊值与低精度上花费大量工程工作。本章把数学定义、线程协作和数据生命周期放在同一条叙述中；先明确前向语义，再讨论反向、量化与融合。

9.1 Softmax：稳定公式需要有效元素契约

对一行 D 个元素，令 $\mathcal{V}$ 为有效索引集合。正常路径要求 $\mathcal{V} \neq \varnothing$，且有效 logits 为有限实数。稳定的 masked Softmax 定义为

$$m = \max_{j \in \mathcal{V}} x_j, \qquad \ell = \sum_{j \in \mathcal{V}} e^{x_j - m}, \tag{9.1}$$

$$y_i = \begin{cases} e^{x_i - m}/\ell, & i \in \mathcal{V}, \\ 0, & i \notin \mathcal{V}. \end{cases} \tag{9.2}$$

减去最大值不改变比值，因为分子分母同时乘了 e^{-m}。它使指数的输入非正，避免对大正数直接求指数造成溢出。比如 $(1000, 1001, 999)$ 转为 $(-1, 0, -2)$，其概率与原式在实数运算下相同。低于浮点范围的很小概率仍可能下溢为零；这与大指数溢出的故障不同。

mask 具有两种含义，必须分清：边界 mask 保护实际内存访问，语义 mask 排除不参与归一化的元素。把越界地址处的加载结果事后覆盖成 $-\infty$，不能补救此前已经发生的非法加载。正确实现须先谓词保护加载，再把无效元素作为 max 的中性值，sum 阶段贡献零。分配给归约的线程仍可全部参与 collective，只让无数据线程贡献中性值。

适用条件与易错点

全 mask 行不是普通稳定公式的合法输入。若无条件令 $m = -\infty$，再计算 $e^{(-\infty)-(-\infty)}$，就会产生 NaN。本书教学契约对空有效集合输出全零，并在 Attention 中同步定义该行输出为零；生产算子必须与调用方约定一致。该约定是对空行的扩展，不是声称空集合存在概率分布。实现应先归约有效计数或判断是否出现有限有效值，再选择全零分支；参与同一 CTA 屏障的线程仍须统一执行同步。

有效输入含 NaN、$+\infty$ 或全部为 $-\infty$ 时，普通减最大值也不足以定义期望结果。本章示例不把它们默默当作有限数处理：正常内核契约为有效值有限，测试显式报告非有限输入；若接口选择传播 NaN，或将若干 $+\infty$ 的概率均分，必须另写分支并与参考实现核对。mask 的 $-\infty$ 是内部排除标记，不等于允许任意非有限有效输入。

9.1.1 从三个阶段到一次驻留

朴素稳定实现先读一遍找 max，再读一遍求 sum，第三遍读入并写概率。按每元素 b 字节计，忽略行统计量约为 $4bD$ 的逻辑流量。若先把指数写入临时数组，流量还要增加，具体数字取决于真正的数据流。对能驻留的短行，每线程把若干元素留在寄存器：归约 max 后直接用原寄存器求指数，归约 sum 后写出，可逼近一读一写的 $2bD$。收益是假设寄存器足以保留这些元素；行很长时，该策略可能 spill，反而引入更多流量。

常见映射是一行交给一个 warp 或一个 CTA，线程以 stride 覆盖列。行数很少且单行很长时，单 CTA 可能不足以利用 GPU，可先按片计算 (m, ℓ)，再合并并归一化。此时多 kernel、临时存储与额外读入也是成本，不能只数指数函数调用。小行的优化则可能主要受 launch 和行打包方式影响，不能把长行最优配置直接移植过去。Triton 的 fused Softmax 教程展示了块级归约与幂次 padding 的一种实现路径，读者仍需核对实际后端生成的资源使用。[14]

9.2 在线归一化：合并的是带参考尺度的状态

对一组有限有效值 $\mathcal{A}$，定义状态

$$S_{\mathcal{A}} = (m_{\mathcal{A}}, \ell_{\mathcal{A}}), \qquad \ell_{\mathcal{A}} = \sum_{i \in \mathcal{A}} e^{x_i - m_{\mathcal{A}}}.$$

两组数据的 max 不同，不能直接相加两个 ℓ。先把它们换到共同尺度 $m = \max(m_{\mathcal{A}}, m_{\mathcal{B}})$，得到

$$\ell = e^{m_{\mathcal{A}} - m} \ell_{\mathcal{A}} + e^{m_{\mathcal{B}} - m} \ell_{\mathcal{B}}. \tag{9.3}$$

这就是在线 normalizer 的关键：每次改变参考最大值，旧分母必须按同样比例缩放。[15]

空集合是特殊的单位状态，应带显式有效标志或计数。可把其存储写成 $(m = -\infty, \ell = 0, \text{valid} = \text{false})$，但合并时若某一侧为空就直接返回另一侧；两侧都空仍返回空。只有两侧非空时才计算式 (9.3)。这样既适用于首个 tile，也适用于 causal mask 使连续若干 tile 全空的情况。浮点合并顺序会影响舍入，数学上的可结合不表示逐位相同。

在线归一化允许一次读输入就得到 max 与分母，但若最终所有概率都要写出，通常仍需再读 logits 或保留它们。它最有价值的场合，是下游只需概率加权后的结果，从而可以把概率的消费也融合进去。

9.3 LayerNorm、Welford 与 RMSNorm

LayerNorm 对一行使用总体方差，即分母为 D：

$$\mu = \frac{1}{D} \sum_j x_j, \qquad \sigma^2 = \frac{1}{D} \sum_j (x_j - \mu)^2, \tag{9.4}$$

$$y_j = \gamma_j (x_j - \mu)(\sigma^2 + \epsilon)^{-1/2} + \beta_j. \tag{9.5}$$

$D > 0$、$\epsilon > 0$ 和归一化的轴是接口契约。方差不是统计学中分母为 $D - 1$ 的无偏估计；换分母会改变算子。

计算 $\frac{1}{D} \sum x_j^2 - \mu^2$ 只需两种简单和，但当输入都接近一个大常数时，两项相减会丢失有效位。Welford 维护 (n, μ, M_2)：

$$n' = n + 1, \qquad \delta = x - \mu, \tag{9.6}$$

$$\mu' = \mu + \delta / n', \qquad M_2' = M_2 + \delta (x - \mu'). \tag{9.7}$$

为在 GPU 上并行，需要的不是串行地把每个元素交给一个线程，而是合并线程局部统计。两组非空状态的合并为

$$n = n_a + n_b, \qquad \delta = \mu_b - \mu_a, \tag{9.8}$$

$$\mu = \mu_a + \delta \frac{n_b}{n}, \qquad M_2 = M_{2,a} + M_{2,b} + \delta^2 \frac{n_a n_b}{n}. \tag{9.9}$$

它来自把 $x - \mu$ 拆为各组内偏差与均值差，并利用组内偏差和为零。空状态直接返回另一组，避免除以零。最后 $\sigma^2 = M_2 / D$。计数乘法应先转换到足够宽的浮点或整数类型，避免 $n_a n_b$ 在窄整数里先溢出；均值、M_2 和累加通常至少使用 FP32。

Welford 增加了状态与合并算术，不能声称它对所有形状都比两遍中心化求方差更快。两遍算法也可数值稳定，只是可能多读一次输入。实际选择需同时约束精度、输入范围、寄存器与吞吐。

RMSNorm 去掉均值中心化，令

$$r = \left(\frac{1}{D}\sum_j x_j^2 + \epsilon\right)^{-1/2}, \qquad y_j = \gamma_j x_j r. \tag{9.10}$$

它减少了统计量，但与 LayerNorm 是不同的数学算子，不能作为无条件等价替换。对 $x = (1, 2, 3)$ 且暂取 $\epsilon = 0$ 的教学手算，LayerNorm 的中心化分子为 $(-1, 0, 1)$，RMSNorm 的分子仍为 $(1, 2, 3)$，分母分别为 $\sqrt{2/3}$ 与 $\sqrt{14/3}$。本书实现测试仍须使用正 ϵ。[16]

“FP32 累加”也不是无限动态范围的保证。BF16 的有限大值平方仍可能超出 FP32 范围。若接口要覆盖这类输入，可用最大绝对值缩放后的平方和，或明确限制输入范围；此时还要正确缩放 ϵ，不能简单把数据除小却保持原式不变。对全零输入，$r = 1/\sqrt{\epsilon}$，输出仍为零。

9.3.1 融合 residual、bias 与 cast 的真实成本

以 $z = x + u$、$y = \mathrm{RMSNorm}(z)$ 为例，分离实现先把 z 写到全局内存再读回；融合可在生成 z 后立即用于统计并写 y。但若训练反向或其他分支还需要 z，它可能仍须物化；这时省掉的只是某些读取和 launch。短行可把 z 留在寄存器，长行则可能需重读。加入 gamma、bias、输出 cast 或量化 scale 时，必须重新计算活跃状态，检查是否出现 spill。可证伪的收益假设应写成“减少了哪一个张量的哪一次读写”，并用资源和流量核对。

9.4 在线 Attention：分母和分子一起换尺度

单头 Attention 为

$$O = \mathrm{softmax}(QK^\top/\sqrt{d} + M)V. \tag{9.11}$$

对一个查询行，用 s_j 表示加入 mask 后的有效分数，v_j 是值向量。若已经处理集合 $\mathcal{A}$，维护

$$m = \max_{j\in\mathcal{A}} s_j, \qquad \ell = \sum_{j\in\mathcal{A}} e^{s_j - m}, \qquad u = \sum_{j\in\mathcal{A}} e^{s_j - m} v_j.$$

读入一个非空新 tile，其 max 为 m_b，取 $m' = \max(m, m_b)$ 和 $\alpha = e^{m-m'}$，更新

$$p_j = e^{s_j - m'}, \tag{9.12}$$

$$\ell' = \alpha\ell + \sum_{j\in b} p_j, \tag{9.13}$$

$$u' = \alpha u + \sum_{j\in b} p_j v_j. \tag{9.14}$$

最后输出 u/ℓ。旧分子也必须乘 α，否则前后 tile 的权重来自不同尺度。[17, 18]

```
# Mathematical pseudocode; one query row, finite valid scores.
valid = False
m, ell, u = -inf, 0.0, zeros(dv)
for tile in key_value_tiles:
    indices = valid_indices_for_this_query(tile)
    if len(indices) == 0:
        continue
    scores = compute_scores(indices)
    mb = max(scores)
    if not valid:
        mnew, alpha = mb, 0.0
    else:
        mnew = max(m, mb)
        alpha = exp(m - mnew)
```

```
    p = exp(scores - mnew)
    ell = alpha * ell + sum(p)
    u = alpha * u + sum(p[:, None] * values(indices), axis=0)
    m, valid = mnew, True
out = u / ell if valid else zeros(dv)
```

这段是行级数学伪代码；映射成 CTA 时不能让空行分支跳过其他行仍需的 CTA collective。实际实现还须控制矩阵片段布局、shared 生产消费、Tensor Core 对齐和 causal mask 的 tile 边界。它故意把首个非空 tile 单独处理，以避免 $-\infty-(-\infty)$；若用条件选择表达式，编译器或语言是否会先求两侧值也需确认。

可核对的算例。三个分数为 $(\log 2, 0, \log 3)$，对应值向量为 $(1,2),(3,0),(-1,4)$。先处理前两个：$m=\log 2, \ell=3/2, u=(5/2,2)$。第三个使最大值变成 $\log 3$，旧状态乘 $2/3$，得到 $\ell'=2, u'=(2/3,16/3)$，所以 $O=(1/3,8/3)$。直接按权重 $(2,1,3)/6$ 求加权和得到同一结果。这个例子检验的是更新公式；GPU 上仍需用多种 tile 划分、mask 与高精度参考检验舍入。

9.4.1 数据生命周期比公式行数更能解释收益

朴素实现把 $N_q \times N_k$ 个分数与概率写到显存。在线版本让一个 Q tile 在片上停留，逐步读取 K/V tile，只保留当前分数块、每行 (m,ℓ) 与输出累加器。它避免了全局物化二次规模的分数矩阵，并不消除一般稠密 Attention 的 $O(N_qN_kd)$ 乘加。输入与输出容量仍为线性规模，片上 scratch 还依赖 tile 大小；如果用户要求返回完整 attention map，该矩阵的二次存储不能凭融合消失。[17]

Q tile 更大可复用更多 K/V，但输出累加器约按 B_Qd_v 增长；K/V tile 更大能减少循环和调度，却增大共享与分数块状态。长序列更可能从避免 HBM 往返中受益，极短序列则可能被 launch、边缘与流水成本主导。FlashAttention-2 对并行与工作划分的改善，应被理解为具体调度和非矩阵算术开销的调整，不能从论文标题推断任意自写在线实现都有同样吞吐。[18]

9.5 Prefill 与 Decode：同一公式的两种数据流

prefill 一次有较多查询，Q 与 K/V 的矩阵块常可形成较充分的乘加并行；decode 每请求通常只有一个或少量新查询，需要扫描已有 KV。若每个 KV 元素仅使用一次，decode 的字节成本就可能比乘加吞吐更关键，增大 batch 或共享 KV 的查询分组可改变复用程度。

仅计某层的 KV 数据，batch 为 B、有效长度为 L、KV 头数为 H_{kv}、每头 K/V 宽度同为 d、元素字节为 b 时，容量为

$$B_{KV}=2BLH_{kv}db.$$

这是布局和 dtype 的字节模型，不包括页表、padding、scale 或分配碎片。GQA/MQA 的多个查询头可以共享 KV，但实现是否眞正复用而非重复读取，需要查流量。

分页 KV 把逻辑 token t 映射为页号 $p=\lfloor t/P\rfloor$ 与页内偏移 $o=t \bmod P$，再用页表找物理地址。页表边界、最后一页的有效长度、请求槽位复用与跨 stream 生命周期都属于正确性。不要用一条“连续 `[B,H,L,D]`”索引假装支持任意分页输入。其收益是假设减少扩容搬移与碎片成本，代价是间接寻址和局部性变化；单算子的延迟与系统请求吞吐要分别评测。

9.6 反向传播与重算：保存的是足够信息

Softmax 的上游梯度为 g 时，有

$$\frac{\partial L}{\partial x_i}=y_i\left(g_i-\sum_j g_jy_j\right). \tag{9.15}$$

只要有概率 y 就不需保存完整 Jacobian。RMSNorm 若 $h_i = g_i\gamma_i$，则

$$\frac{\partial L}{\partial x_i} = rh_i - \frac{x_i r^3}{D}\sum_j h_j x_j. \tag{9.16}$$

式 (9.16) 说明前向保存输入和每行 r 可以换取较少反向重算，但 γ 的梯度还涉及跨行归约。训练实现不能只通过前向测试就声称算子完整；应做小尺寸双精度有限差分或框架梯度检查，并报告低精度误差。

在线 Attention 反向常保存每行的 log-sum-exp $m + \log \ell$，再按 tile 重算概率，以算术换二次中间张量的容量。若前向有 dropout，还须复用相同的随机 mask。用 keep probability p 定义 $z_i = m_i x_i/p$、$m_i \sim \mathrm{Bernoulli}(p)$，其随机数应由 seed 与稳定的逻辑元素 counter 对应，而不是由 CTA 实际调度顺序决定。改变 tile 后若 counter 到逻辑元素的映射也改变，就不能自动保证逐位重算一致。全 mask 行与 padding 也需有一致约定。

9.7 量化：表示、scale 与累加是一个算法

对称 INT8 的一个明确约定是

$$q = \mathrm{clip}_{[-127,127]}\left(\mathrm{round}_{\mathrm{even}}(x/s)\right), \qquad \hat{x} = sq, \quad s > 0. \tag{9.17}$$

这里故意不使用 -128，以保持正负对称；其他库可能使用不同范围。若用最大绝对值定 scale，则非全零组取 $s = a_{\max}/127$，全零组约定 $s = 1$，避免除零。非饱和区间的舍入误差至多约 $s/2$，饱和区间不能用该界；NaN/Inf 的转换策略必须另定。

仿射量化 $\hat{x} = s_x(q_x - z_x)$ 带零点，点积展开为

$$\sum_k \hat{a}_k\hat{b}_k = s_a s_b\Big(\sum_k q_{a,k}q_{b,k} - z_b\sum_k q_{a,k} - z_a\sum_k q_{b,k} + Kz_a z_b\Big). \tag{9.18}$$

漏掉零点修正不是小的精度误差，而是实现了另一个表达式。若 scale 沿 K 分组，不同组通常要分别缩放后求和，不能把一个统一 scale 提到整个整数点积外。INT32 累加也要检查最坏范围；当 K 很大时，其上界约按 $Kq_{a,\max}q_{b,\max}$ 增长。

FP8 的指数范围缓解固定整数刻度的一些问题，却仍需要 scale、舍入、饱和与 amax 策略。当前张量的精确 amax 通常需要先遍历再量化；延迟 scale 可复用历史统计来减少额外同步或遍历，但遇到分布突变可能增加饱和。per-tensor、per-channel 与分块 scale 依次增加元数据和索引工作，并可能降低离群值对整组的影响。论文中的 E4M3/E5M2 定义不等于任意框架类型的字节编码，必须记录确切格式。[13]

FP4、NVFP4、MX 等更低精度路径还涉及 block scale、打包顺序和特定 MMA 支持。本书不把“4 位”当作统一格式，也不在没有匹配 GPU、Toolkit 与库 descriptor 的情况下宣称它可直接运行。低位权重可能减小读取量，但 unpack、scale 加载、反量化与形状 padding 也可能主导小矩阵。适用边界应通过精度、字节量、指令和端到端时间共同判断。[19, 11]

9.8 稀疏与 MoE：零值不是自动省下的工作

2:4 结构化稀疏路径要求被压缩的四元素组满足相应保留数约束，并携带合法 metadata；沿哪一维分组、如何排列与压缩依赖具体指令和库。把稠密矩阵的一半元素设为零，再调用稠密 GEMM，不会自动执行稀疏 Tensor Core。压缩、布局转换与小形状的开销也应包含在相应计时范围。[12, 11]

MoE 的局部实现往往包含 token 分桶、gather、分组 GEMM 与 scatter/reduction。专家负载不均时，一张漂亮的大 GEMM 性能表无法解释小专家和路由开销。先用真实或明确标为合成的路由分布，记录每专家 token 数、容量约束、重复路由的累加语义和端到端时间；多 GPU 的通信依赖见第 十四 章。本书聚焦算子的数据流，不在这里展开模型架构或集群容量设计。

9.9 练习：让边界输入参与设计

1. 实现有限值 masked Softmax 的 CPU 参考，测试空行、全 mask、单个有效值、$(1000, 1001, 999)$ 与不规则行长。提示：分别检查和为一、mask 为零以及空行的约定；随机输入通过不足以覆盖 $-\infty$ 的分支。
2. 将同一行分成若干块，以不同顺序合并 (m, ℓ)；再给每个元素加相同常数。提示：在容差内验证平移不变性，比较舍入而不要求逐位相同。
3. 对近常数大偏置输入比较平方和相减、两遍中心化与 Welford。提示：使用独立 FP64 参考；输入转成 FP32 时已损失的差异，任何后续算法都不能恢复。
4. 重做本章 Attention 手算，并在第一块和中间块加入全 mask tile。提示：检查有效标志、旧分子缩放和最终空行分支；不要通过把 $-\infty$ 换为任意大负数掩盖语义问题。
5. 为对称 INT8 写打包与反量化测试，加入全零、离群值、正负舍入中点与超过校准范围的输入。提示：单独报告饱和率，不能只看均方误差。

这些算子的共同经验是：融合减少了中间存储，却把原来由 kernel 边界保证的依赖移到了寄存器、共享内存和 collective 中。下一章将把同样的问题扩展到 stream 与主机调度。

第十章

异步执行：把依赖、生命周期与时间线对齐

一个 GPU kernel 可以完全正确，组合进程序后却偶尔读到旧数据；也可能单独很快，端到端运行时一直等待主机。原因常在 kernel 之外：提交顺序不等于跨 stream 的先后，主机返回不等于设备完成，缓冲区的生命期也不等于某个函数的作用域。本章从依赖图出发组织 streams、events、双缓冲与 CUDA Graphs。这里的重叠都是有条件的执行机会，需要第 六 章的时间线证据。[1, 2]

10.1 三个时刻：提交、开始、完成

调用 `kernel<<<...,stream>>>` 通常把工作提交给设备后返回。对某项操作，应区分主机调用完成、设备操作开始以及设备操作完成三个时刻。普通主机变量的程序顺序只能约束 API 提交过程；另一个 stream 中的消费者，必须由 CUDA 依赖显式连接到生产者。

同一 stream 的工作按其顺序语义执行，后续消费者可依赖前面的写入；不同 stream 的独立操作则可能重叠，也可能因资源、调度或隐式同步而串行。stream 不是专用 SM 的所有权：两个 stream 不能保证各占一半 GPU；已经占满寄存器或共享内存的 kernel 可能没有资源让另一个 kernel 并发驻留。

默认 stream 有不同模式。legacy default stream 会与某些其他 stream 发生隐式同步；per-thread default stream 的语义不同。显式创建 `cudaStreamNonBlocking` stream 可避免与 legacy default stream 的那类隐式同步，但不消除真实数据依赖。实验须记录编译和运行的默认 stream 模式，库则应遵从调用者传入的 stream，或框架当前设备上的 current stream。[1]

10.2 Event 是依赖边，不只是秒表

设 A 在 stream s_p 产生数据，B 在 s_c 消费。最小依赖为

$$A \longrightarrow \text{record}(\text{ready}, s_p) \longrightarrow \text{wait}(s_c, \text{ready}) \longrightarrow B.$$

用禁用计时的 event 表达依赖，避免不必要的时间戳功能。下面是主机调度片段，所有分配、输入填充和错误检查宏已由调用方准备；缓冲区在末尾同步前均保持存活：

```
cudaStream_t producer, consumer;
cudaEvent_t ready, consumed;
CUDA_CHECK(cudaStreamCreateWithFlags(&producer, cudaStreamNonBlocking));
CUDA_CHECK(cudaStreamCreateWithFlags(&consumer, cudaStreamNonBlocking));
CUDA_CHECK(cudaEventCreateWithFlags(&ready, cudaEventDisableTiming));
CUDA_CHECK(cudaEventCreateWithFlags(&consumed, cudaEventDisableTiming));

CUDA_CHECK(cudaMemcpyAsync(d_x, h_x_pinned, bytes,
                           cudaMemcpyHostToDevice, producer));
kernel_a<<<grid, block, 0, producer>>>(d_x);
CUDA_CHECK(cudaPeekAtLastError());
CUDA_CHECK(cudaEventRecord(ready, producer));
CUDA_CHECK(cudaStreamWaitEvent(consumer, ready, 0));
kernel_b<<<grid, block, 0, consumer>>>(d_x, d_y);
CUDA_CHECK(cudaPeekAtLastError());
CUDA_CHECK(cudaEventRecord(consumed, consumer));

// d_x may be overwritten only after kernel_b has finished reading it.
CUDA_CHECK(cudaStreamWaitEvent(producer, consumed, 0));
```

```
// Any later reuse of d_x on producer is now ordered safely.

CUDA_CHECK(cudaStreamSynchronize(consumer));
CUDA_CHECK(cudaStreamSynchronize(producer));
CUDA_CHECK(cudaEventDestroy(consumed));
CUDA_CHECK(cudaEventDestroy(ready));
CUDA_CHECK(cudaStreamDestroy(consumer));
CUDA_CHECK(cudaStreamDestroy(producer));
```

`cudaStreamWaitEvent` 建立设备侧等待，通常不阻塞主机直到 event 完成；`cudaEventSynchronize` 则让主机等待。前者用于 GPU 内流水，后者用于主机真的要读取、修改或释放相关数据时。event 记录的是提交到对应位置以前的工作，不能自动覆盖另一个 stream 中没有连到它的活动。

同一个 event 可以重新 record，但等待对应的是该 wait 提交时关联的最近一次记录。复杂流水若让不同线程或不同阶段无规则地复用一个 event，容易等待到错误的一代。初学实现为每个缓冲槽位分配明确的 ready/done 事件，并在日志中标注代数；不要把“最后调用 record 的是它”当作跨主机线程的同步协议。调用未记录过的 event 也不能代替等待未来将要提交的工作。[1]

10.3 资源生命期：什么时候才能复用地址

每个缓冲都应回答“最后一个使用者是谁”。异步 H2D 提交后，主机不能立刻修改或释放源 pinned 内存；异步 D2H 提交后，主机不能立刻读目的内存。设备输入在最后一个消费者结束前不能被下一批覆盖；输出在最后一个读取者结束前不能被回收。函数返回时局部 C++ 包装器被析构，可能比设备消费结束更早。

page-locked（pinned）主机内存为 H2D/D2H 与计算重叠提供重要条件，通常用 `cudaMallocHost` 或 `cudaHostAlloc` 分配。pageable 内存的 API 可能通过内部 staging，主机调用是否阻塞取决于方向与实现；因此不能以函数名带 `Async` 就保证重叠，也不能把全部 RAM 页锁定当作免费优化。分配和页锁定应移出稳态循环，测量时仍需验证实际时间线。[2]

`cudaMallocAsync/cudaFreeAsync` 把分配与释放接入 stream 顺序，并不让裸指针跨 stream 自动安全。若 buffer 在 s_1 分配、在 s_2 使用，就需为分配就绪和最终释放分别表达依赖。PyTorch 的缓存分配器另有自己的 stream 记录机制；转到额外 stream 的 Tensor 需要等待生产者，并在必要时 `record_stream` 防止存储被过早复用。`record_stream` 管理分配器生命期，不是数据已经就绪的 event。[20]

10.4 双缓冲：两个槽位是一张状态机

将输入划分为 L 个 chunk，每个槽位包含 pinned 输入、pinned 输出、设备输入、设备输出和一个 stream。一个槽位的合法状态是

$$\text{主机可写} \to \text{H2D} \to \text{计算} \to \text{D2H} \to \text{主机可读} \to \text{再次可写}.$$

双缓冲让槽位 $t \bmod 2$ 处理 chunk t，另一个处理相邻 chunk。最简单的安全实现以每槽位的 done event 保护主机复用，单槽位内的设备顺序由 stream 保证：

```
// Scheduling pseudocode: two preallocated slots; CUDA calls are checked.
bool busy[2] = {false, false};
for (size_t chunk = 0; chunk < chunks; ++chunk) {
  int slot = int(chunk % 2);
  if (busy[slot]) {
    cudaEventSynchronize(done[slot]);
    consume_output(h_out[slot], previous_chunk[slot]);
  }
  size_t count = fill_input(h_in[slot], chunk);
  cudaMemcpyAsync(d_in[slot], h_in[slot], count * sizeof(float),
                  cudaMemcpyHostToDevice, stream[slot]);
```

```
  launch_chunk(d_in[slot], d_out[slot], count, stream[slot]);
  cudaMemcpyAsync(h_out[slot], d_out[slot], count * sizeof(float),
                  cudaMemcpyDeviceToHost, stream[slot]);
  cudaEventRecord(done[slot], stream[slot]);
  previous_chunk[slot] = chunk;
  busy[slot] = true;
}
for (int slot = 0; slot < 2; ++slot) {
  if (busy[slot]) {
    cudaEventSynchronize(done[slot]);
    consume_output(h_out[slot], previous_chunk[slot]);
  }
}
```

这个写法在重新填输入前也先消费旧输出，避免两类主机复用错误。非整除的最后 chunk 单独传入有效长度，不能总是搬运和计算满槽位容量。若 `fill_input` 或 `consume_output` 很慢，主机本身也会限制流水；用更多槽位能否改善，应由时间线判断，而不是随意增加并发。

若单个 chunk 的 H2D、计算、D2H 时间分别为 T_H, T_C, T_D，理想的独立阶段流水可近似为

$$T_{\text{pipe}} \gtrsim T_H + T_C + T_D + (L-1)\max(T_H, T_C, T_D). \tag{10.1}$$

这是充分缓冲、独立资源、无额外争用的教学模型。设备若只有一条相关拷贝资源，或两个方向共享瓶颈，应合并相应阶段，例如以 $T_H + T_D$ 估计拷贝服务负载；两个槽位也不保证达到三阶段理想极限。拷贝与 kernel 都可能争用显存，重叠之后各自时长可能增加，不能把独立测得的时间当成不变常数。

对这一优化，正确的实验问题是：“在相同总输入、chunk 划分、结果消费与分配范围下，双缓冲是否降低端到端时间，且时间线显示目标阶段确实重叠？”只报告某一个 kernel 的 event 时间，无法回答这个问题。

10.5 CUDA Graphs：复用的是工作图与提交结构

当许多短 kernel 和拷贝按稳定拓扑重复执行时，逐次主机提交可能占可见成本。CUDA Graph 把操作节点与依赖边保存为可重复 launch 的执行图。capture 是构图方式之一，不是让任意 Python 或 C++ 控制流在 GPU 上自动执行。[1]

```
// stream, buffers and library state are prepared and warmed up first.
cudaGraph_t graph = nullptr;
cudaGraphExec_t exec = nullptr;
CUDA_CHECK(cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal));
kernel_a<<<grid, block, 0, stream>>>(d_x, d_tmp);
kernel_b<<<grid, block, 0, stream>>>(d_tmp, d_y);
CUDA_CHECK(cudaStreamEndCapture(stream, &graph));
CUDA_CHECK(cudaGraphInstantiate(&exec, graph, nullptr, nullptr, 0));
for (int step = 0; step < steps; ++step)
  CUDA_CHECK(cudaGraphLaunch(exec, stream));
CUDA_CHECK(cudaStreamSynchronize(stream));
CUDA_CHECK(cudaGraphExecDestroy(exec));
CUDA_CHECK(cudaGraphDestroy(graph));
```

该片段示范的是固定指针、固定拓扑的设备操作。调用者须检查 capture 中的 launch 和结束返回值，失败后按文档恢复 stream 状态。普通主机计算不会因位于 capture 范围而每次 replay；捕获期间若主机读回数据决定下一次分支，这个过程不能按原样变成固定图。

使用前应逐项检查：捕获的库操作是否支持当前版本的 capture；首次初始化、JIT 与必要 warm-up 是否已结束；Tensor 或裸指针指向的存储在整个 replay 期间是否有效；随机数和更新状态是否有正确的 replay 语义；shape 或地址变化能否通过受支持的节点参数或 graph exec update 处理。并非所有分配都一律禁止：支持的 stream-ordered allocation 可有图内语义，但普通可能同步的 API 不能随意混入 capture，应以所用 CUDA 版本的图支持规则为准。

多 stream capture 还需要通过事件把支路从捕获主流分出，并在结束前按文档汇合；创建了两个 stream 并不表示它们都会自动进入同一张图。capture mode 控制某些潜在不安全 API 的约束范围，也不是关闭所有依赖检查的性能开关。

Graph 的收益是假设重复提交开销占比足够大、构图与实例化能被足够次数的 replay 摊薄。若稳态每轮节省 $\Delta T > 0$，一次构图/实例化增加成本 T_I，粗略盈亏次数为 $T_I/\Delta T$。这是规划式，实际数值应分开测量。长 GEMM 已主导的程序、频繁 shape 改变的路径或内存生命周期不稳定的服务，可能收益很小。Graph 本身不改变 kernel 的数学 FLOPs 或保证提高其 Tensor Core 利用率。

10.6 结构化协作：CTA、cluster 与 cooperative grid

`cooperative_groups` 通过明确的 group 表达 CTA 或 tiled subgroup，减少手工管理参与者的歧义。所有 group collective 都需要该 group 约定的成员参与；给变量起名 `group` 不能解除第 三 章的同步约束。

cooperative grid 的全局同步依赖特定 launch 方式、设备支持与驻留限制。若普通 kernel 的所有 CTA 试图用全局标志自旋等待，而一部分 CTA 尚未被调度，已驻留 CTA 就可能阻止生产者获得资源。应优先用 kernel 边界或明确受支持的 cooperative launch，不把全局 atomic counter 当作通用 grid barrier。

Hopper 引入 thread-block cluster 与 distributed shared memory（DSM）。cluster 的 CTA 可以在受支持范围协作，但并非整个 grid 的任意同步域。使用远端 CTA 的共享内存前要确保所有拥有者已启动并完成初始化；拥有者退出之前要确保远端访问已结束，因此需相应的 cluster 同步与生命期协议。cluster 尺寸、调度与 occupancy 需用对应设备和 launch 属性核对，较大的协作范围也可能降低独立可调度任务数。[21, 1]

10.7 练习：从故障时序倒推缺失的边

1. 在两 stream 的生产消费代码中，先删除 ready wait，再删除 consumed wait，分别说明可能读到什么数据。提示：第一种是读写依赖，第二种是下一代覆盖旧输入；小测试偶尔正确不能证明没有竞争。
2. 给双缓冲增加第三个槽位并测试非整除总长度。提示：将有效长度、chunk 编号和 done 事件全部归属到槽位，不能用最后一轮的全局变量解释全部输出。
3. 设计一个覆盖所有支路的端到端计时边界。提示：在一个计时 stream 中放 start，其他流等 start；末尾各支路 record done，计时流等待全部 done 后放 stop，或用主机时间包住明确的总完成边界。单流 stop 不会自动等待无依赖的另一流。
4. 比较 eager 提交与 Graph replay，分别计入和排除一次构图成本。提示：固定指针、输入与执行次数，验证每次 replay 的输出；不要把没有重新装填输入的旧结果当成新请求。

异步优化的第一项交付应是一张无环依赖图，每条边对应真正的顺序原语，每块内存标明最后使用者。性能证据则用时间线说明这些正确的独立部分是否真的重叠。

第十一章

PyTorch 扩展与 Triton：从 kernel 到框架

本章的学习目标

本章说明一个自写 kernel 如何进入真实框架：设备、dtype、stride、current stream、Tensor 生命周期和自动求导都属于算子契约。CUDA C++、Triton、CUB/CUTLASS 与框架编译器各有边界，选择应由缺口和证据决定，而不是由语言偏好决定。

11.1 先定位调用链和缺口

典型调用链为

$$\text{Python} \to \text{Dispatcher/ATen} \to \text{CUDA 或 Triton} \to \text{PTX} \to \text{SASS}.$$

先以 cuBLASLt、cuDNN、CUB、CUTLASS、FlashAttention 或 PyTorch 原生实现建立强基线；只有在需要特殊布局、融合、量化或库尚未覆盖的算子时，才进入 custom op。CUDA C++ 适合精确控制同步、指令和资源；Triton 适合规则块级算子的快速迭代；CUTLASS/CuTe 适合可组合 GEMM 和 epilogue；Inductor 等图编译器适合跨算子调度。每条路径都要回到同一套正确性和计时协议。[20, 22, 10, 11, 14]

11.2 C++/CUDA extension 的最小契约

以下骨架省略构建文件，但保留生产代码最容易遗漏的边界检查和 current stream：

```cpp
#include <torch/extension.h>
#include <ATen/cuda/CUDAContext.h>
#include <c10/cuda/CUDAGuard.h>

__global__ void saxpy_k(const float* x, const float* y, float* z,
                        float a, int64_t n) {
  for (int64_t i = int64_t(blockIdx.x) * blockDim.x + threadIdx.x;
       i < n; i += int64_t(blockDim.x) * gridDim.x)
    z[i] = fmaf(a, x[i], y[i]);
}

at::Tensor saxpy_cuda(const at::Tensor& x, const at::Tensor& y, double a) {
  TORCH_CHECK(x.is_cuda() && y.is_cuda(), "CUDA tensors required");
  TORCH_CHECK(x.scalar_type() == at::kFloat && y.scalar_type() == at::kFloat,
              "float tensors required");
  TORCH_CHECK(x.sizes() == y.sizes(), "shape mismatch");
  TORCH_CHECK(x.is_contiguous() && y.is_contiguous(),
              "this teaching kernel requires contiguous input");
  c10::cuda::CUDAGuard guard(x.device());
  auto out = at::empty_like(y);
  auto stream = at::cuda::getCurrentCUDAStream();
  int64_t n = x.numel();
  int block = 256;
  int grid = int((n + block - 1) / block);
  saxpy_k<<<grid, block, 0, stream.stream()>>>(
      x.data_ptr<float>(), y.data_ptr<float>(), out.data_ptr<float>(),
      float(a), n);
  C10_CUDA_KERNEL_LAUNCH_CHECK();
  return out;
}
```

```
31
32 TORCH_LIBRARY(myops, m) { m.def("saxpy(Tensor x, Tensor y, float a) -> Tensor"); }
33 TORCH_LIBRARY_IMPL(myops, CUDA, m) { m.impl("saxpy", &saxpy_cuda); }
```

真实扩展还要决定是否接受非 contiguous 输入，或显式转换并把转换时间纳入端到端测试；用 `CUDAGuard` 绑定正确设备，并使用调用者的 current stream。不在库 kernel 中调用 `cudaDeviceSynchronize()`，也不能让裸 device 指针逃离 Tensor 生命周期。

若算子需要反向传播，应注册 Autograd 实现或用 Python 的 `torch.autograd.Function` 明确保存量和重算成本。要配合 `torch.compile`，还需提供 fake/meta 实现、shape 推断并检查 graph break。

11.3 Triton 的块级视角

```
1 @triton.jit
2 def saxpy_tl(x_ptr, y_ptr, z_ptr, a, n, BLOCK: tl.constexpr):
3     pid = tl.program_id(axis=0)
4     offs = pid * BLOCK + tl.arange(0, BLOCK)
5     mask = offs < n
6     x = tl.load(x_ptr + offs, mask=mask, other=0.0)
7     y = tl.load(y_ptr + offs, mask=mask, other=0.0)
8     tl.store(z_ptr + offs, a * x + y, mask=mask)
```

一个 Triton program 可以粗略看成可向量化的块；`tl.arange` 形成 lane，mask 同时保护边界，`tl.sum` 和 `tl.dot` 表达块内归约与矩阵乘。`autotune` 可以搜索 BLOCK、`num_warps` 和 `num_stages`，但 key 必须包含真正影响资源和数据流的维度。跨 program 的归约仍需 kernel 阶段或框架调度，Triton program 不是隐含的全局同步域。

概念	CUDA C++	Triton
坐标	`threadIdx/blockIdx`	`program_id+tl.arange`
tile	显式 shared、寄存器与 barrier	编译器管理的块级布局
调优	手动选择 block、tile、寄存器	autotune 搜索候选配置
跨块同步	kernel 边界或受限 cooperative launch	通常拆成多个 program/kernel

11.4 何时使用更高层工具

CuPy/Numba 适合快速原型；Thrust/CUB 适合 map、sort、scan 等成熟组合；Triton 适合规则融合；CUTLASS/CuTe 适合 GEMM、Tensor Core、epilogue 和量化；手写 CUDA 适合特殊异步搬运、WGMMA、精确同步或跨架构 dispatch。高层抽象降低开发成本，也可能引入临时分配、编译时间、动态 shape 限制或无法访问的指令路径。选择时保留一个可解释 baseline，并用生成的 IR/PTX、资源和时间验证实际路径。

11.5 框架集成的易错点

常见失败包括：默认 stream 与调用者 stream 不一致；非 contiguous 输入被当作连续数组；autotune key 漏掉 stride 或 dtype；Tensor 在异步工作完成前被复用；算子注册缺少 CPU、Meta 或 Autograd 路径；`torch.compile` 因 unsupported op 发生 graph break。先让错误在最小 shape 上确定性出现，再用 Compute Sanitizer 和框架的同步调试选项定位；不要用一次偶然的同步让 race “消失”。

11.6 第十一章检查清单与练习

- 能说明 custom op 的 device、dtype、shape、stride、stream 和生命周期契约。
- 能解释 CUDA C++、Triton、CUB/CUTLASS 与图编译器的适用边界。
- 能从生成的 IR/PTX 和 profiler 证实框架实际调用了哪条路径。
- 能为一个需要 backward 的算子列出保存、重算和 graph break 的代价。

练习：将随书 SAXPY 改成一个 PyTorch custom op 和一个 Triton 版本；对 contiguous、转置视图、零长度和非整齐长度分别做参考校验，并把 Python 调度、kernel、同步和端到端时间分开记录。

第十二章

现代 GPU 架构与可移植特化

本章的学习目标

本章建立“可移植基线 + 条件 fast path + 运行时 dispatch”的架构策略。产品名、架构名、compute capability、PTX 和 SASS 处在不同层次；异步搬运和 Tensor Core 只有在对应设备、Toolkit、布局和同步协议同时满足时才是有效优化。

12.1 先查 capability，再写 fast path

运行时应查询设备属性，编译时为目标 compute capability 生成代码，并保留确实需要的 PTX fallback。示意代码如下：

```
cudaDeviceProp prop{};
CUDA_CHECK(cudaGetDeviceProperties(&prop, device));
if (prop.major >= 9 && supports_required_feature(prop))
  launch_newer_path(args);
else
  launch_portable_path(args);
```

不要仅凭产品名称猜测指令或共享内存容量；同一代内部也可能有不同 SKU。发布记录至少包含 GPU、compute capability、driver、Toolkit、host compiler、编译目标和实际选中的 dispatch 分支。[1, 23, 3]

12.2 Ampere、Hopper 与 Blackwell 的差异

代际（示例）	常见目标	与 kernel 相关的能力和限制
Ampere（A100/GA100）	sm_80；另有 sm_86	TF32、BF16/FP16/INT8 Tensor Core、部分 SKU 的 2:4 稀疏、`cp.async` 与异步 barrier。GA100 与 GA10x 的资源和能力不能互换。
Hopper（H100/H200）	sm_90	FP8、TMA 多维异步搬运、`mbarrier`、thread-block cluster/DSM 和 WGMMA；tensor map、cluster 形状与布局高度特化。
Blackwell（多个 SKU）	依 Toolkit 与 SKU 而定	新的 Tensor Core、block-scaled FP4/NVFP4 路径及延续的 TMA/cluster 能力；dtype、指令和共享内存上限必须查对应版本文档。

架构表不是性能数字。是否更快仍取决于 shape、布局、stage 数、寄存器和实际瓶颈；对小矩阵，特化路径的启动和边界成本可能超过收益。[12, 21, 19]

12.3 异步搬运的完成语义

Ampere 的 `cp.async` 可以把 global-to-shared 搬运从显式寄存器路径中分离；Hopper 的 TMA 以 tensor map 描述多维 bulk copy，并用 `mbarrier` 通知消费者。概念流水为：提交阶段 $t+1$ 的搬运，等待阶段 t 已完成，再让计算读取阶段 t 的 shared tile。提交、完成和可见性是三件事，缺少 wait 或 barrier 会产生 correctness bug。

```
// Pseudocode: exact pipeline APIs depend on CUDA Toolkit and target SM.
for (int stage = 0; stage < STAGES; ++stage) {
  async_copy(global + next, shared + next, bytes);
  commit_or_arrive(stage);
```

```
  wait_until_ready(current);
  consume(shared + current);
  current = (current + 1) % STAGES;
}
```

stage 数增加会提高可隐藏的搬运深度，也会增加 shared、barrier 状态和填充/排空时间。应在边界 tile、短 K 和低并发 shape 上单独测试，不能把流水图上的重叠当作已实现的吞吐。

12.4 PTX、SASS 与生成代码核验

PTX 是虚拟 ISA，SASS 是目标 GPU 的机器指令；PTX 中出现某条指令不等于最终 SASS 仍保留它。可以用：

```
nvcc -O3 -lineinfo -Xptxas=-v -arch=sm_90 -cubin kernel.cu -o kernel.cubin
nvdisasm kernel.cubin
cuobjdump --dump-ptx --dump-sass executable
```

看到 `mma.sync`、`wgmma`、`ldmatrix` 或异步搬运相关指令，只能证明路径被选中；还需结合 issue slot、依赖链、内存 stall 和正确性结果判断收益。用编译器报告检查寄存器和 spill，用 Nsight Compute 检查实际 pipe 利用率。[4, 9]

适用条件与易错点

不要把一个架构上可编译的内建函数写成所有机器都能执行的公共接口。用条件编译和运行时能力检查保护特化代码，并让 fallback 保持相同的 shape、dtype、误差和同步契约。

12.5 第十二章检查清单与练习

- 能区分产品名、架构名、compute capability、PTX 与 SASS。
- 能为一个 tile kernel 设计 portable baseline、fast path 和 dispatch 条件。
- 能解释异步 copy、commit、wait、barrier 与消费者读取之间的依赖。
- 能用 `cuobjdump`/`nvdisasm` 和 Nsight 验证生成指令及资源。

练习：为同一 GEMM tile 编译一个基线目标和一个架构特化目标；比较编译资源、SASS、正确性和端到端时间，写出哪些收益来自布局/流水，哪些只在特定 compute capability 上成立。没有 NVIDIA GPU 时完成静态检查，并把运行项标为待验证。

第四部分

留下证据：调试、通信与研究实验

第十三章

调试、工具链与可复核证据

本章的学习目标

本章把“结果偶尔不对”或“似乎更快”变成一组可重做的检查。读完后，应能定位错误发生在哪一个阶段，理解检测工具的覆盖边界，并从一次分析得到下一项可以证伪的实验。

13.1 先定义算子的输入契约

调试从一个能够失败的断言开始。为算子写下形状、数据类型、行列主序、允许的 stride、地址对齐和输入输出是否可以别名；规定空输入、越界位置以及 NaN/Inf 的语义。以转置为例，本书算例只接受连续行主序输入，输出独立分配，不支持原地转置；这比用一句“适用于矩阵”更有用。若扩展到带 stride 的张量，地址计算和测试都必须随之改变。

数学参考应独立于优化路径。GEMM 的 CPU 参考以同一组 FP32 输入进行 double 点积，不能另生成一组 double 输入；转置只搬运数值，可要求逐元素完全一致；归约改变加法顺序，应按数值尺度和归约深度给出误差预算。接近零的参考值使纯相对误差失去意义，常用逐元素规则是

$$|\hat{y}_i - y_i| \le \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}}|y_i|.$$

任何出现的非有限数值应按输入契约处理，不能让 NaN 比较返回 false 后悄悄通过检查。若采用“所有输出都必须有限”的契约，应先单独检测有限性，再比较误差。

边界矩阵至少包含空输出、一个元素、warp 和 block 邻界前后、tile 边界前后、细长矩形以及 K=0。再补输入契约内的极端值、零、重复值和抵消求和。GPU 指针的乘积索引要及时提升为 `size_t`；把已经溢出的 32 位乘积强转为 64 位无法补救。任意输入接口还要检查分配字节乘法溢出。

13.2 把错误绑定到具体阶段

观察到的失败	首先检查的证据
编译或链接失败	Toolkit 与 host 编译器兼容性、头文件、设备链接、目标架构、符号签名。
launch 后立刻报错	block/grid 维度、动态共享内存、kernel 参数和该设备是否有兼容代码。
同步或拷贝时才报错	更早的异步访问越界、非法地址、设备断言与生命周期；该同步点未必是根因。
重复运行偶尔出错	未初始化读、缺少生产者到消费者依赖、错误参与 mask、复用尚未完成的缓冲。
正确但突然变慢	不同编译选项、spill、输入布局、频率、并发进程、隐藏同步与路径回退。

异步 API 的成功返回通常只表示工作已被接受。launch 后检查配置错误，在明确的 stream 边界检查执行错误：

```
kernel<<<grid, block, shared_bytes, stream>>>(args);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaStreamSynchronize(stream));
```

`cudaGetLastError()` 返回并清除调用线程的 runtime last-error 状态；`cudaPeekAtLastError()` 返回但不清除。它们都不能代替检查每个 API 的直接返回值，也不是等待执行完成的操作。清除状态本身

不会“删除”一个已经保存并报告的错误；真正的问题是忽略返回值，让后续代码失去最早失败的位置。某些 API 还可能返回先前异步工作的错误，因此日志应记录操作、输入形状、设备、stream 和首个失败返回。出现严重设备异常后，通常应终止该测试进程并重新运行，不能假设调用一个清错函数就恢复了有效上下文。有关错误传播与执行模型见官方指南 [1]。

用 `CUDA_LAUNCH_BLOCKING=1` 可让部分异步错误更靠近触发点，但它改变了时序。`-G` 用于设备调试，`-lineinfo` 用于保留源码行对应；二者不是同一类开关。调试构建、设备 printf、Sanitizer 和分析器中的时间都不能直接作为正常性能结果。

13.3 一条有先后关系的工具链

先把输入缩小到仍能复现错误的尺寸，随后按以下顺序排查 [24, 3]：

1. 使用正常 release 构建跑独立参考和断言；保存 seed 与最小输入。
2. 跑 memcheck，先修非法访问与对齐问题。否则后续竞态报告可能只是越界的次生现象。
3. 跑 initcheck、racecheck、synccheck，逐项保存退出码和报告。`--error-exitcode` 让发现错误时自动化任务确实失败。
4. 正确性稳定后，用 `-O3 -lineinfo -Xptxas=-v` 保存资源报告，再进行正常计时。
5. 用 Systems 看依赖与空洞，用 Compute 看一个热点 kernel 的资源和吞吐，最后用 SASS 核对生成指令。

```
# Run from manuscript/examples after run_gpu.sh has built the binaries.
GPU_DIR=../../output/experiments/kernel_cuda_frontier_ai-jingfeng/gpu
compute-sanitizer --error-exitcode=99 --tool memcheck \
  "$GPU_DIR/basic_kernels" --small
compute-sanitizer --error-exitcode=99 --tool racecheck \
  "$GPU_DIR/basic_kernels" --small
bash profile_gpu.sh
```

适用条件与易错点

racecheck 主要报告共享内存访问的危害，不能视作全部全局内存、跨 stream 或跨 GPU 竞态的证明器。initcheck 的全局内存初始化检查、synccheck 的同步用法检查也各有覆盖范围。所有工具无报告，只能说明“这些输入和执行路径未被该工具检出问题”，不能代替对共享状态生命周期的推理。具体支持的指令与检查范围随工具版本变化。

例如双缓冲的数据在两个阶段间交替复用，应分别证明“加载完成后才读取”和“所有读取完成后才覆盖”。同一个 ready event 只表达一个方向，不能自动证明两个方向。测试除小尺寸外，还要覆盖尾 tile、不同 pipeline 深度与重复调用；尤其不能用删掉异步路径的“调试版本”证明异步路径正确。

13.4 Nsight 任务应从问题开始

Systems 与 Compute 回答的问题不同 [8, 9]。如果端到端慢而 kernel 本身很短，先看 Systems 的 CPU API、GPU stream、传输和空洞；如果一个稳定热点占了大部分 GPU 时间，再用 Compute 分析它。两种工具都需要记录版本、参数、输入与原始报告。

以 naive 与 shared-tiled GEMM 为例，可提出假设：“tile 重用减少对 A/B 的重复全局加载，但新增屏障和共享内存访问。”先固定 $M = 257, N = 129, K = 67$、FP32、相同输入和编译选项，在无分析器的进程中保存两者的 median/p95。再执行本书 `profile_gpu.sh`；它使用该固定形状，分别过滤两个 kernel，并保存本机可用 metric 名称。检查：

1. Systems 是否显示意外拷贝、同步或 launch 空洞；两个版本是否执行了相同工作量。
2. Compute 中全局 load 的流量、SM 与内存吞吐是否支持“减少数据移动”的解释；观察共享内存访问和屏障代价，而非只看单个命中率。
3. 资源报告中寄存器、shared 和 local 是否变化。local 字节不是运行时 spill 次数，occupancy API 的 active blocks/SM 也不是实测吞吐。
4. SASS 中是否存在预期的浮点 FMA、load/store 与 barrier。源码用了 `fmaf` 并不意味着使用了 Tensor Core。

Compute 可能通过 replay 多次执行 kernel；缓存、频率与测量过程也会受影响。因此分析器中的 harness 时间不能与无分析器时间拼接为加速比。计数器名称和可用权限依架构、驱动及工具版本而变，先用 `ncu --query-metrics` 确认，不要把某个架构的指标名当作跨设备常数。

13.5 一次运行应留下什么

最小证据包需要源码版本或哈希、编译命令、硬件及驱动、Toolkit 和 host 编译器、输入形状/布局/dtype、seed、误差规则、预热与采样次数、计时区间、每个样本和统计定义。还应记录功耗/频率策略、并发任务和 MIG/MPS 配置；这些条件未固定时，要如实说明。不同版本的比较必须保持数值任务相同；允许误差增大或更换 dtype 是另一组实验。

本书的 GEMM 程序把 CUDA Events 的单次 kernel 区间与 host steady clock 的传输完整区间分开。后者包含 pinned H2D、launch、D2H 和 stream 完成同步，明确排除分配和 CPU 参考；真实应用还含布局转换或内存分配时，必须另测相应场景。小 kernel 的 event 时间受分辨率与调度影响，可另加批量重复实验，但应说明它测量的是一批 launch，而不是擅自把测量范围变了还保留原标签。

13.6 本章练习与提示

1. 在副本中删去分块 GEMM 内循环末尾的屏障。说明为何下一个 tile 的写可能覆盖其他 warp 尚在读的内容，再比较普通运行和 racecheck 的表现。**提示：**无报错不能反驳这个数据依赖；改变尺寸和 block 调度，记录失败路径，不修改正式基线。
2. 转置采用 NaN sentinel 初始化输出后要求逐位相同。为什么这比重复使用上一次正确输出更容易发现漏写？**提示：**写入覆盖性是与数值精度不同的断言。
3. 为一个“快了但 max relative error 很大”的归约诊断报告。**提示：**先看参考和是否接近零，再报告绝对误差与 $\sum_i |x_i|$，并检查输入与累加 dtype。

- 失败日志能指向输入契约、操作阶段与首个已检查的错误。
- 工具结论带有版本、输入、覆盖范围与原始报告。
- 数值误差、资源、kernel 时间与端到端时间各有明确含义。
- 每个性能解释都对应可重做的比较和一个下一步实验。

第十四章

多 GPU：依赖、通信与计算重叠

本章的学习目标

多 GPU 将单设备的“谁写、谁读、何时能复用”扩展到多设备和多进程。这里关注算子的跨设备数据流与测量，不展开集群运维。本章的完成标准是画清依赖并测得正确结果与 step 时间，而不是仅让两条时间线看起来重叠。

14.1 先确认地址、设备和可用路径

每个 host 线程都有当前 CUDA 设备状态。分配、创建 stream/event、launch 和释放之前，必须清楚它们属于哪张卡；一个指针的数值不能代替设备归属。程序应记录设备数量、可见 GPU 映射、计算能力和内存容量，再查询每对设备的 peer access 能力。`CUDA_VISIBLE_DEVICES` 会改变进程内编号，日志中的逻辑 0 未必是机器上物理编号 0。

UVA（统一虚拟寻址）统一了部分地址空间的命名规则；它不意味着任意 GPU 都能直接访问任意指针，也不保证该访问有高带宽。P2P 能力具有方向性，要对两个方向分别查询，按使用场景启用 peer access 并检查返回值。支持性、路由和速度是三个问题，不能由一个 capability 查询同时回答 [1]。

```
nvidia-smi -q
nvidia-smi topo -m
nvcc --version
```

将这些记录与实际传输实验一起保存。PCIe 层级、NUMA、NVLink/NVSwitch、GPU 实例配置和并发任务都会影响路径。不能仅由“同型号两张卡”推出相同通信代价。`cudaMemcpyPeerAsync` 的名字中有 Async，也不能推出在当前拓扑、缓冲位置与调度条件下必然与计算重叠。

14.2 集合通信的正确性先于算法选择

All-reduce 将每个 rank 的输入按同一归约操作组合，并使每个 rank 得到完整结果。rank 是通信域内的参与者编号，不等同于 host 线程编号。NCCL 的 API 和流语义应以实际安装版本的用户指南为准 [25]。

一次正确的 collective 要求所有参与者对通信域、操作顺序、元素数、dtype 与归约操作达成一致。某一 rank 因条件分支跳过一个 bucket，其他 rank 可能永久等待；同一块地址在一端当 FP32、另一端当 FP16，错误也不会由“函数名相同”消除。跨 rank 的 shape 元数据应在进入热点循环之前核对。输入、输出与通信域必须活到设备工作真正完成。

NCCL 可以选择 ring、tree 或其他算法、协议和拓扑路径。ring 的常用理想通信模型中，每个 rank 的 all-reduce 传输量约为

$$2\frac{P-1}{P}S,$$

其中 P 是 rank 数，S 是每个 rank 的张量字节数。这个模型假定标准 reduce-scatter 加 all-gather，不是所有实际算法的精确流量公式，更不是延迟保证。小消息可能受启动延迟影响，大消息更可能暴露持续带宽限制；实际选择应看安装版本与拓扑上的测量。

测试 all-reduce 时，可令 rank r 的每个元素为 $r+1$。求和参考是 $P(P+1)/2$，可直接检查所有输出；随后再加入随机输入和浮点容差。只测总时间而不检验每个 rank 的输出，会漏掉 dtype、count 和缓冲复用问题。

14.3 一个 bucket 的依赖关系

设 compute stream 生成梯度 bucket G_j，communication stream 做 all-reduce，优化器随后消费归约结果。需要两条独立依赖：

$$\text{produce}(G_j) \to E_j^{\text{ready}} \to \text{allreduce}(G_j) \to E_j^{\text{done}} \to \text{consume}(G_j).$$

第一条防止通信读到尚未写完的梯度，第二条防止优化器读到尚未归约完成的数据。与此同时，compute stream 可以计算不依赖 G_j 的其他工作。所有资源复用还必须等到最后的消费者结束；done event 不自动覆盖优化器的后续使用。

以下是**单进程管理多个 GPU 的调用顺序片段**，不是完整可编译程序。它假定每个 rank 已创建默认阻塞初始化的 NCCL communicator、归属正确的 compute/comm stream 和禁用计时的 ready/done events，所有 API 使用检查宏，且独立工作访问不同缓冲。

```
// ready[], done[], compute[], comm_stream[], nccl_comm[] belong to each GPU.
for (int r=0; r<P; ++r) {
  CUDA_CHECK(cudaSetDevice(device[r]));
  produce<<<grid, block, 0, compute[r]>>>(grad[r], count);
  CUDA_CHECK(cudaGetLastError());
  CUDA_CHECK(cudaEventRecord(ready[r], compute[r]));
  CUDA_CHECK(cudaStreamWaitEvent(comm_stream[r], ready[r], 0));
}
NCCL_CHECK(ncclGroupStart());
for (int r=0; r<P; ++r) {
  CUDA_CHECK(cudaSetDevice(device[r]));
  NCCL_CHECK(ncclAllReduce(grad[r], grad[r], count, ncclFloat,
                           ncclSum, nccl_comm[r], comm_stream[r]));
}
NCCL_CHECK(ncclGroupEnd());
for (int r=0; r<P; ++r) {
  CUDA_CHECK(cudaSetDevice(device[r]));
  CUDA_CHECK(cudaEventRecord(done[r], comm_stream[r]));
  independent_work<<<grid2, block, 0, compute[r]>>>(other[r]);
  CUDA_CHECK(cudaGetLastError());
  CUDA_CHECK(cudaStreamWaitEvent(compute[r], done[r], 0));
  consume<<<grid, block, 0, compute[r]>>>(grad[r], output[r], count);
  CUDA_CHECK(cudaGetLastError());
}
```

同一进程向多个 GPU 发起相关 NCCL 操作时，group 可避免 host 在一张设备的调用上等待其他设备尚未发出的操作。`ncclGroupEnd()` 返回并不表示 GPU 通信已完成；完成关系由 stream/event 表达。若显式采用非阻塞 communicator，group 还可能处于 in-progress 状态，必须按该版本 NCCL 的状态查询流程确认提交状态，再发依赖它的 CUDA 操作；不能原样套用上面的阻塞初始化假设。

多进程模式通常每个进程只管理一个 rank，但所有进程仍须以一致顺序发 collective。检查同步 API 的错误返回之外，还要按版本支持监测异步 NCCL 错误，采用应用超时并使所有 rank 一起退出失败运行。超时只能揭示停滞，不能说明一定是网络问题。

14.4 为什么已经异步仍然没有收益

设独立计算耗时 T_c，通信耗时 T_m。忽略启动、竞争及依赖限制，理想重叠下的局部下界为

$$T_{\text{serial}} = T_c + T_m, \qquad T_{\text{overlap}} \geq \max(T_c, T_m).$$

实际还要加不可重叠的前后缀、launch、event 与排队开销。NCCL 通信 kernel 也使用 GPU 执行与内存资源；它可能减慢原来的计算。于是时间线有交叠，step 时间仍可能上升。报告必须同时给出串行、重叠与实际 workload 的关键路径时间。

bucket 太小会增加消息数、host 提交与协议启动代价，太大则等到较晚才准备好，压缩重叠窗口。双缓冲或更多 in-flight bucket 增加显存与生命周期复杂度。测试时逐项改变 bucket 字节数，保持总通信量与总计算任务不变；先验证每个 bucket，再观察关键路径。对临界长尾，应保存 rank 的延迟分布和最慢 rank，而非仅平均值。

常见故障按依赖顺序定位：

1. 生产者尚未完成：检查 ready 是否记录在真正写入数据的 stream 上，而非 host 以为的“主 stream”。
2. 通信尚未提交：检查 rank 是否都进入相同 collective、group 结束和非阻塞状态。
3. 消费者过早读取：检查 done 事件和消费 stream 的 wait，而非只在通信 stream 内排顺序。
4. 下一步覆盖缓冲：检查最后一次使用，必要时增加双缓冲及对应完成事件。
5. 正确但无重叠：检查默认 stream 语义、host 同步、资源饱和、bucket 就绪时刻和共享带宽竞争。

14.5 怎样测得一个可信的多 GPU 结论

先分别测通信独立运行、计算独立运行与串行组合，再测同一总工作量的重叠组合。预热、输入、stream 设置和数值检查保持一致。一次完整 step 应在所有 rank 完成后才停止计时；多进程场景报告各 rank 的本地时间与最大值，并说明开始协调方式。跨设备或跨机器的 event 时间戳不能直接相减；CUDA Events 的 elapsed-time 区间应位于同一设备。Systems 的关联时间线用于检查调度关系，正常无插桩运行用于性能统计。

若报告有效吞吐，写明分子是应用 payload、总字节量，还是按算法模型折算的 bus bandwidth。all-reduce 的算法带宽与上述 ring 模型下的折算总线带宽不是同一个数。峰值链路带宽也不能直接等同于实际有效吞吐。

MIG 划分可见资源和隔离边界，MPS 改变多进程共享 GPU 的执行方式；它们均需按目标设备与版本确认可用性，并记录配置。不要把 MPS 理解为“让多个进程共享同一个 CUDA context”，也不要假定 MIG 实例具备完整 GPU 同样的 P2P 路径。没有多 GPU 环境时，本章的实现与测量方案保持待验证，不能由单卡或 CPU 结果填入扩展效率。

14.6 本章练习与提示

1. 为上述一个 bucket 的片段画出两个 stream 的顺序图，标出可以重叠的 independent work。**提示：**先连 ready/done，再把梯度缓冲最后一次使用到下一次覆盖之间的边连上。
2. 在两 GPU 上令每个 rank 输入不同常量，比较无 overlap 和有 overlap 的输出。随后扫 bucket 大小。**提示：**保持总 payload 不变，把改变消息数引起的开销一并报告。
3. 某 rank 提前返回造成其他 rank 超时，应如何缩小故障范围？**提示：**保存每个 rank 最后成功进入的 collective 编号、count、dtype、设备和错误，而非只保存 rank 0 的日志。

- 每个指针、stream、event 和 communicator 的设备归属清楚。
- 所有 rank 的 collective 元数据与顺序一致，生命周期覆盖真实完成。
- 正确性、通信独立时间、计算独立时间与 step 时间分别有证据。
- 收益按关键路径和数值契约判断，异步与时间线重叠不充当完成证明。

第十五章

实验路线：从正确结果到优化解释

本章的学习目标

项目的产物是能够运行的算子、独立参考与一份可以重做的报告。优化之前先提出收益假设，优化之后同时检查数值、数据移动、资源与完整任务时间。本章给出本书实际交付的起点，并区分已执行的验证与尚待 GPU 的工作。

15.1 把路线划成三个可验收阶段

原路线以 P0 到 P12 编号，包含环境探针在内共十三项，而不是“十二级”。新版保留这些能力，但按依赖分成三个阶段，避免读者在索引与误差尚未掌握时直接堆叠架构特化。

阶段一：把索引和同步写对。完成环境探针、SAXPY、转置、归约以及 scan/histogram。SAXPY 要求 grid-stride 覆盖所有元素，转置要求尾 tile 的共享内存读有对应初始化，归约要求所有参与线程到达屏障。scan 还需明确 exclusive/inclusive、运算单位元与跨 block 前缀的传播。验收先看独立参考、空/奇数/非整齐尺寸，再看有效字节数与时间。

阶段二：把数据复用和精度联系起来。从 FP32 naive GEMM 到 shared tile，再到 register tile 或库级 Tensor Core 路径；实现稳定 Softmax、RMSNorm 与在线 Attention。每增加一次融合或更改累加格式，就重新检查中间量、尾部掩码和资源上界。Tensor Core 比较须保留输入格式与误差预算，并将 CUDA core FP32 基线、TF32 与 FP16/BF16 场景分别报告。

阶段三：把一个 kernel 放回应用。接入框架的当前 stream、检查 stride/alias 与 autograd 契约，再比较 Triton/CUDA/高性能库的实际路径。随后加入 graphs、异步搬运和 NCCL；先写出完整依赖图，再用 step 时间验证收益。框架扩展不自动继承普通 C++ 程序的内存生命周期规则，应按版本核对调用和流语义 [20]。

对照任务	要证伪的假设	至少保存的证据
转置 pad0/pad1	padding 降低列读的 shared bank 冲突	同协议时间、地址映射、冲突指标。
naive/tiled GEMM	重用减少全局加载，收益超过屏障代价	误差、流量、寄存器/shared、时间。
独立/融合 norm	减少写回胜过寄存器与并行度代价	中间字节、spill、长短行时间。
显式/在线 Attention	分块状态保持输出并降低中间存储	高精度参考、峰值显存、长短序列。
串行/overlap 通信	缩短关键路径胜过资源竞争	所有 rank 结果、时间线、step 分布。

15.2 随书代码可以怎样开始

`manuscript/examples/` 中两个 CUDA C++17 程序分别有自己的 main：

- `basic_kernels.cu` 包含 SAXPY、两阶段归约、朴素转置，以及 32×32 tile 配 32×8 线程块的无

padding 和加一列 padding 转置。所有版本都有 CPU 参考和误差断言，转置输出先以 sentinel 填充，防止漏写被旧内容掩盖。

- `gemm_progression.cu` 在同一组 FP32 A/B 上比较 naive 与 shared tile=16 的 GEMM，包含空输出和 K=0、非 tile 倍数与不同长宽比。它分别保存 CUDA Events kernel 样本与传输完整区间样本、median、nearest-rank p95 和资源信息。

共享的 `common.cuh` 负责错误检查、资源管理和 JSON 输出。它是教学 harness，不是替代生产框架的完整内存管理库。最小构建入口位于同目录的 `run_gpu.sh`；先核对 Toolkit、host 编译器和目标 GPU，再显式选择支持的架构：

```
cd manuscript/examples
# Example only: select the architecture supported by your GPU and Toolkit.
CUDA_ARCH=sm_80 bash run_gpu.sh
```

脚本保存完整编译选项与 `ptxas` 输出，不默认使用 fast math。若 nvcc 或 NVIDIA 驱动探针不可用，脚本记录 not_run 并退出。编译报错、误差超标或运行错误也会使脚本失败；不能把部分 JSONL 当成整项测试通过。

15.3 渐进 GEMM 的公平对照

第一轮仅改变 A/B 的共享内存复用：两版保持相同输入、FP32 FMA 累加、输出定义、线程几何、编译选项和计时协议。CPU 参考使用同一 FP32 数值做 double 点积，误差断言同时考虑绝对与相对误差。输入值被限制在有界区间，给定容差只对该教学测试矩阵建立覆盖，不是任意模型的数值保证。

第二轮增加 register tile 时，先写资源预算。若一个线程持有 $r_m r_n$ 个 FP32 累加器，仅累加器就至少需要相应数量的 32 位寄存器值，还未计地址、加载缓冲与临时量。每个输出增加复用的同时，可能降低驻留 block 数、增加 local 访问或放大尾部浪费。假设“更高算术强度”应由减少的流量和最终时间支撑，不能只由源码循环结构支撑。

第三轮调用高性能库或 Tensor Core 后，应分别报告更换计算路径带来的精度/格式变化。库基线要包含实际 layout、workspace、算法选择与 epilogue；不能将自己的融合算子与不含同样后处理的库 GEMM 混称为同一任务。对不同 batch、细长矩阵、小 K 和尾部严重的矩阵分别画出或列出结果；一个大方阵的收益不能代表全部输入。

随书 harness 的 kernel 区间使用同一 stream 的 start/stop events，输入已在设备驻留；host 端的另一组测量包含 pinned H2D、launch、D2H 和完成同步，排除分配、数据生成与参考计算。默认每版预热 5 次、采样 31 次，保留原始样本。固定执行顺序便于教学复现，但正式性能比较还应交错或随机版本顺序、重复多个进程并记录频率、并发和缓存条件，避免把热身或温度漂移当成优化收益[2]。

15.4 本次实际执行的证据与边界

本次修订宿主为 macOS arm64，没有 nvcc 或可用 NVIDIA GPU。已实际执行的是 Python 3 标准库的数学与索引契约验证，seed 为 20261021；命令、Python/平台版本、源码 SHA-256、具体输入和结果在

`output/experiments/kernel_cuda_frontier_ai-jingfeng/`

中保存。该目录的 CPU 报告与 GPU preflight 分开，读者可按同样命令复做：

```
python3 cpu_verify.py --output \
  ../../output/experiments/kernel_cuda_frontier_ai-jingfeng/cpu
```

CPU 验证覆盖五组性质：SAXPY 对高精度 Decimal 的标量误差；两阶段归约对独立 `math.fsum` 的误差预算及每元素恰被访问一次；转置所有有效 store 都读取已加载的共享地址，并构成正确的双射；GEMM 尾 tile 与 K=0 对独立点积参考；在线 Softmax/Attention 在不同分块、长度及整体偏移下对高精度指数参考的一致性。另行验证全部屏蔽行出现 $-\infty-(-\infty)$ 的未定义数值，要求实现显式选择该行的语义。

这些结果支持数学递推和地址覆盖；不能证明真实 CUDA 编译、设备屏障行为、生成指令、低精度硬件舍入、GPU 资源计数或性能。GPU 编译、全部 GPU 正确性、四类 Sanitizer、Nsight/SASS、Tensor Core 与多 GPU 运行均列为待验证，正文不附本次未测得的吞吐或加速比。后续可用 `profile_gpu.sh` 执行分析并保存报告。

15.5 一页报告的内容

好的报告由一条推理链组织，而不是按工具截图排列：

1. **任务与假设：**写清输入范围、数值契约、要减少的工作量和可能代价。
2. **独立参考：**解释参考与被测实现的关系，列出边界/异常输入和误差规则。
3. **单一改动：**保留 naive、中间版和当前版，使每一步优化可单独归因。
4. **可比证据：**固定输入与计时范围，保存全部时间样本、资源、流量与必要的汇编证据。
5. **退化范围：**说明哪些 shape、精度、设备或并发条件使结果变差，并给出原因或待检验解释。
6. **下一步：**提出一个能推翻当前解释的具体实验，例如“固定 tile 改 padding，看冲突计数下降而时间是否改变”。

若用论文报告数字作为背景，要标明论文、硬件、版本与工作负载；若用合成数据解释 Roofline，要写“教学数据”；若尚未运行，应写“实验预期”。这三类信息不能和本次实测放进同一张无标记的性能表。强基线、数值结果与端到端收益同样重要：局部 kernel 变快但完整 step 不变，是需要解释的有效实验结论。

15.6 本章练习与提示

1. 从随书 GEMM 添加一个 register tile 版本，预先写下寄存器与流量假设。**提示：**先保持输入和 FMA 语义，再在所有原有边界上回归；别只复制一个更大的 block。
2. 将在线 Attention 的 CPU 验证扩展为部分 mask、极大 score 和一行全部 mask。**提示：**普通有限行的数学等价性不能规定全部 mask 行的业务语义，先明确接口再断言。
3. 设计一个融合能降低 kernel 时间却恶化端到端时间的可测场景。**提示：**考虑额外布局转换、workspace、launch 数与动态 shape，保持任务等价。

- 代码、命令、输入、版本和原始结果能独立定位，原始归档不被覆盖。
- 已执行、待验证、模型推断与文献结论的标签清楚。
- 优化含正确性与退化案例，性能结论有范围与数值约束。
- 学习终点是能解释完整算子的数据流、同步、误差与时间。

附录 A

接口速查：先核对契约

速查表用于回忆，不代替对应 Toolkit 的接口说明。每项 API 都需要回答三件事：哪些线程或队列参与、完成条件是什么、结果对谁可见。编译器接受一个调用，并不表示这三个条件已经满足。

A.1 线程、地址与资源

接口或语法	使用前提与检查
`__global__`	设备 kernel；主机 launch 异步提交，返回类型为 void。
`__host__ / __device__`	双重限定函数的两条编译路径都必须合法，不能无条件调用另一侧专用 API。
`threadIdx / blockIdx`	软件坐标不保证 block 执行顺序。二维 block 的线性线程号为 $t_x + B_x(t_y + B_y t_z)$。
`size_t / int64_t`	乘法前提升类型；事后把溢出的 32 位结果赋给 64 位变量不能补救。还要检查分配字节数是否溢出。
`__restrict__`	别名契约，不能用于实际重叠而按独立缓冲区优化的指针。
`cudaFuncGetAttributes`	查询已编译 kernel 的寄存器、静态共享内存等；动态共享内存另计。
`cudaOccupancyMaxActive BlocksPerMultiprocessor`	给定 block 大小和动态共享内存估计驻留上限；不是最佳性能配置搜索器。

A.2 同步与内存顺序

原语	能保证什么，以及不能省去什么
`__syncthreads()`	CTA 屏障和该范围内先前内存访问的可见性。活动线程应按合法控制流到达同一屏障；不提供任意跨 CTA 同步。
`__syncwarp(mask)`	指定且参与调用的 warp lane 之间的屏障与内存排序；掩码不是任意数据集合的压缩编号。
`__ballot_sync`	对已知参与集合执行投票。逻辑成员集合应在分歧之前确定，不能靠分歧后的瞬时活跃集合倒推。
`__shfl_down_sync`	参与 lane 的寄存器交换。源 lane 必须存在并参与；它不替代共享内存生产者和消费者之间的内存屏障。
`__threadfence()`	调用线程的设备范围内存排序；不让其他线程等待，不单独建立安全的发布与获取协议。
`atomicAdd`	对指定地址进行原子读改写。传统原子操作不能凭一个地址上的原子性替代其他数据的发布或全局阶段完成。

原语	能保证什么，以及不能省去什么
`cuda::atomic_ref`	显式 memory order 和 scope；发布与获取、缓冲区复用和进度条件仍需算法保证。

warp、fence 和原子语义的正式定义见 Programming Guide 与 PTX 文档 [1, 4]。

A.3 队列、内存与错误检查

接口	契约提示
`cudaMemcpyAsync`	参数、方向与 stream 必须正确。pageable 主机内存可能触发 staging 或主机阻塞；不能由函数名推出重叠。
`cudaMallocHost`	页锁定主机缓冲区帮助 DMA；完成前不得释放或覆盖仍在传输的数据。
`cudaMallocAsync` / `cudaFreeAsync`	分配与释放遵守 stream 顺序。跨 stream 使用必须显式传递依赖，释放不能早于所有使用者。
`cudaEventRecord`	在指定 stream 上记录先前已提交工作的事件节点；记录调用返回不等于该工作已经完成。
`cudaStreamWaitEvent`	给目标 stream 增加依赖，不阻塞主机以等待事件；只能等待正确记录世代的事件。
`cudaPeekAtLastError`	查看当前错误状态，不清除。应结合调用前状态和紧邻 launch 的检查定位，不能保证错误一定来自最后一次 launch。
`cudaGetLastError`	返回并清除当前线程的最后错误状态，不会等待 kernel 完成；保存错误后再处理。
`cudaStreamSynchronize`	等待指定 stream，并检查返回的执行期错误；库函数不应擅自增加设备级全局同步。

A.4 从文档查到可运行环境

按“GPU 型号和 CC、驱动、Toolkit、host compiler、库版本、二进制目标”分别记录环境。CUDA 兼容性文档讨论驱动与应用的边界 [23]，它不会替用户证明某条架构专属指令能在任何后代 GPU 上执行。NVCC 的 SASS 目标、虚拟架构和兼容选项见编译器手册 [3]。设备能力查询和实际编译、运行测试应同时保留；缺一项时就明确其验证状态。

附录 B

符号、单位与手算检查

B.1 统一符号

符号	本书含义
G, B, T	grid 的 block 数、block 的线程数、算法 tile 边长；二维或三维坐标另标下标。
M, N, K	$C_{M\times N} = A_{M\times K}B_{K\times N}$ 的三个逻辑维度。
ld, pitch	leading dimension 常以元素计，CUDA 2D 拷贝的 pitch 以字节计。
m, ℓ, u	在线 Softmax 的行最大值、缩放后指数和、未归一化加权向量。
$\epsilon, u_{\mathrm{fp}}$	数值稳定项与浮点单位舍入误差；两者不能互换。
F, D, I	浮点运算数、指定层级的传输字节数、算术强度 $I = F/D$。

本书以十进制 GB/s 和 GFLOP/s 记录带宽及计算吞吐：1 GB $= 10^9$ 字节，1 GFLOP $= 10^9$ 次浮点运算。容量若用 GiB 则是 2^{30} 字节。一次 FMA 通常按两次 FLOP 计数；这里的计数是算法口径，不等同于所有机器指令的数量。

B.2 用四个算子检查流量口径

以下均为 FP32、连续且非重叠输入，输出只写一次，不计分配和传输，默认 $N, M, K > 0$。这是**逻辑流量模型**；缓存命中、写分配和实际 sector 数必须另测。

$$\begin{aligned} \text{SAXPY:}\quad & F = 2N, \quad D = 12N, \quad I = 1/6; \\ \text{转置:}\quad & D = 8MN; \\ \text{GEMM 理想复用:}\quad & F = 2MNK, \quad D_{\min} = 4(MK + KN + MN). \end{aligned}$$

GEMM 此处采用 $C = AB$，不读取旧 C；若 $C \leftarrow \alpha AB + \beta C$ 且 $\beta \neq 0$，还需计入旧 C 的读取。朴素的逐输出点积在源代码加载计数上约为 $D_{\mathrm{load}} = 8MNK$，加上 $4MN$ 的写入。这不是说 HBM 一定承担相同流量，因为 L1/L2 可以重用输入；它也不是理想的 $D_{\min}$。对方阵，$I_{\min\text{-traffic}} = N/6$，朴素的无重用口径约为 1/4。把这两个值混用会让 Roofline 图失去解释力 [26, 2]。

归约读取 $4N$ 字节只是第一阶段下界。假设第一阶段有 G 个部分和，第二阶段读取并写出结果，那么部分和的额外流量约为 $4G + 4G + 4$ 字节。若分更多层，就继续计入每层的读写，不能把中间缓冲区当作不存在。

B.3 把数值契约写成判定式

对有限参考值 r_i 和计算值 y_i，随书程序采用混合容差：

$$|y_i - r_i| \le \mathrm{atol} + \mathrm{rtol}|r_i|.$$

绝对误差照顾参考接近零的情况，相对误差照顾幅值变化；最大相对误差用于报告时需另外给分母下界。NaN/Inf 不应通过一个带 NaN 的比较悄悄消失：先检查非有限值，再按算子的显式策略判断。容差需要结合累加长度、输入范围和 dtype 确定，不能让所有算法共用一个任意常数。

浮点加法满足舍入误差模型时，长度 n 的顺序求和有常见上界形式

$$|\hat{s} - s| \le \gamma_{n-1} \sum_i |x_i|, \qquad \gamma_k = \frac{k u_{\mathrm{fp}}}{1 - k u_{\mathrm{fp}}}, \quad k u_{\mathrm{fp}} < 1.$$

该式不是任意输入的相对误差保证，尤其在严重抵消时不能由此推出“相对误差很小”。固定归约树使同环境、同实现的顺序稳定，不自动带来跨编译器、跨架构逐位一致。

B.4 术语不互相替代

术语	本书用法
CTA / block	可在块内协作的线程集合；cluster 是另一个受设备和启动约束的协作层级。
SIMT / warp	单指令多线程的执行组织；lane 有独立状态，不能据此假设隐式锁步共享内存通信。
occupancy	活跃 warp 与该 SM 上限的比值；高 occupancy 不等于低延迟，也不等于满吞吐。
HBM / global	HBM 是显存技术之一；global 是编程模型的地址空间，消费级 GPU 也可能使用 GDDR。
TLP / ILP	线程级并行与指令级并行，均不能凭增加线程数无限放大。
TMA / WGMMA	架构特定的数据搬运及 warp-group 矩阵乘机制，不能抽象成所有 GPU 通用的 API。
fence / barrier	内存顺序约束与参与者等待；二者的作用域和语义需要分别检查。
kernel / E2E	事件括住的设备工作区间与明确定义的端到端区间；两者需要各自的基线和计时器。

附录 C

阅读资料与证据分级

C.1 原始资料分别回答什么

Programming Guide 与 PTX ISA 用于核对线程参与、同步、内存顺序、地址空间和指令前提；Best Practices Guide 用于建立可检验的访问和测量模型 [1, 4, 2]。编译器的目标架构、资源报告和生成文件选项以 NVCC 手册为准 [3]。这些材料中的“支持”不等于本书在当地机器上测得了某种吞吐。

对常用并行模式，先读 Blelloch 的 scan 推导，再看 Merrill 和 Garland 的 decoupled look-back，最后检查 CUB 对实际设备、类型和版本的接口 [6, 7, 5]。本书用三阶段 scan 解释依赖；单遍方案及库实现可继续参看 decoupled look-back 与 CUB。

GEMM 的强基线是 cuBLAS/cuBLASLt，定制 tile、epilogue 和布局可研究 CUTLASS/CuTe[10, 11]。编写 PyTorch 算子要同时读 CUDA 语义与 custom operator 教程，而不只照搬一段绑定代码 [20, 22]。Triton 的向量加法教程给出块级 program、mask 与 launch 的具体关系 [14]。

C.2 从数学到算法论文

在线归一化器的更新来自 Milakov 与 Gimelshein 的工作 [15]。FlashAttention 把在线统计与片上分块结合，使稠密精确 Attention 不必把完整分数矩阵写到 HBM；FlashAttention-2 改进并行划分；FlashAttention-3 则针对 Hopper 的异步机制和低精度进行组织 [17, 18, 27]。本版按 arXiv 指定版本核对作者、题名与算法叙述。论文中的 exact 指算法未采用稀疏或低秩近似，不意味着浮点输出与任何物化 Softmax 的实现逐位一致。

RMSNorm 的省略中心化与 FP8 格式的范围、特殊值约定分别见原始资料 [16, 13]。格式定义、scale 策略、目标指令和任务精度是四个不同层次，不能由一个格式名称推导出部署效果。

C.3 工具证据的读取顺序

先使用 Compute Sanitizer 缩小越界和同步错误，再用 Nsight Systems 查看依赖和空隙，用 Nsight Compute 解释单 kernel 的资源与瓶颈 [24, 8, 9]。Nsight 的 replay、序列化和采样会改变时间；因此 profile 与性能基线分开保存。通信的 stream 语义以 NCCL 官方资料为准 [25, 28]，不能因为时间线上出现同时活动的颜色块就断言端到端获得收益。

证据等级	可支持的结论
手算或模型	某种地址映射、依赖或理想流量的推导；不是运行时间。
CPU 数学检查	数学变换、边界覆盖与参考误差在已测输入上成立；不是 CUDA 内存模型的验证。
GPU 正确性运行	指定环境与输入的设备结果、错误检查和 sanitizer 状态；不外推所有形状。
GPU 性能运行	同条件、同数值契约、可追踪计时区间下的统计；不移用论文数字。
应用测量	额外包含调度、传输、框架或通信的端到端收益；必须说明区间。

C.4 如何继续核验

本项目的 `evidence/source-verification.json` 保存地址、读取日期、HTTP 状态和 SHA-256；`evidence/` 中的逐章记录说明读到了哪些段落及它们支持的纠错。页面成功下载本身并不能证明论断，所以正文仅使用已与算法、接口原文或独立推导对照的部分。Roofline 的出版元数据通过 Crossref 核对，出版商正文未能直接取得；本书的操作性解释同时对照 NVIDIA 文档，不声称本次读到了不可取得的全文。

实际命令、种子和输出位于本主题 `output/experiments/`，GPU 项目的空缺以状态记录保留，读者可按算例目录的说明补做。实验失败时应保留失败日志和输入，而不是只替换成最后一次成功运行。当公式、代码、编译器和测量结果能够相互解释，教材才成为继续研究的起点。

参考文献与原始技术资料

[1] NVIDIA. *CUDA C++ Programming Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/cuda-c-programming-guide/index.html (visited on 10/21/2026).

[2] NVIDIA. *CUDA C++ Best Practices Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/cuda-c-best-practices-guide/index.html (visited on 10/21/2026).

[3] NVIDIA. *CUDA Compiler Driver NVCC*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/cuda-compiler-driver-nvcc/index.html (visited on 10/21/2026).

[4] NVIDIA. *Parallel Thread Execution ISA*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/parallel-thread-execution/index.html (visited on 10/21/2026).

[5] NVIDIA. *CUB: Cooperative Primitives for CUDA C++*. CCCL v3.0.0 repository README. URL: https://raw.githubusercontent.com/NVIDIA/cccl/v3.0.0/cub/README.md (visited on 10/21/2026).

[6] G. E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. Technical report metadata checked; algorithm also checked in the author's 1993 chapter. Carnegie Mellon University, 1990. URL: https://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-90-190.html (visited on 10/21/2026).

[7] D. Merrill and M. Garland. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Tech. rep. NVR-2016-002. NVIDIA, 2016. URL: https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back (visited on 10/21/2026).

[8] NVIDIA. *Nsight Systems User Guide*. Online documentation. URL: https://docs.nvidia.com/nsight-systems/UserGuide/index.html (visited on 10/21/2026).

[9] NVIDIA. *Nsight Compute Profiling Guide*. Online documentation. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html (visited on 10/21/2026).

[10] NVIDIA. *cuBLAS Library User Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/cublas/index.html (visited on 10/21/2026).

[11] NVIDIA. *CUTLASS Documentation*. Online documentation. URL: https://docs.nvidia.com/cutlass/latest/index.html (visited on 10/21/2026).

[12] NVIDIA. *NVIDIA Ampere GPU Architecture Tuning Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/ampere-tuning-guide/index.html (visited on 10/21/2026).

[13] P. Micikevicius et al. *FP8 Formats for Deep Learning*. arXiv v2; preprint version checked. 2022. arXiv: 2209.05433. URL: https://arxiv.org/abs/2209.05433v2 (visited on 10/21/2026).

[14] Triton contributors. *Vector Addition: Triton Tutorial*. Online documentation. URL: https://triton-lang.org/main/getting-started/tutorials/01-vector-add.html (visited on 10/21/2026).

[15] M. Milakov and N. Gimelshein. *Online normalizer calculation for softmax*. arXiv v2; preprint version checked. 2018. arXiv: 1805.02867. URL: https://arxiv.org/abs/1805.02867v2 (visited on 10/21/2026).

[16] B. Zhang and R. Sennrich. *Root Mean Square Layer Normalization*. arXiv v1; preprint version checked. 2019. arXiv: 1910.07467. URL: https://arxiv.org/abs/1910.07467v1 (visited on 10/21/2026).

[17] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. arXiv v2; preprint version checked. 2022. arXiv: 2205.14135. URL: https://arxiv.org/abs/2205.14135v2 (visited on 10/21/2026).

[18] T. Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. arXiv v1; preprint version checked. 2023. arXiv: 2307.08691. URL: https://arxiv.org/abs/2307.08691v1 (visited on 10/21/2026).

[19] NVIDIA. *NVIDIA Blackwell Tuning Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/blackwell-tuning-guide/index.html (visited on 10/21/2026).

[20] PyTorch contributors. *CUDA Semantics*. PyTorch 2.14 documentation snapshot. URL: https://docs.pytorch.org/docs/2.14/notes/cuda.html (visited on 10/21/2026).

[21] NVIDIA. *NVIDIA Hopper Tuning Guide*. CUDA Toolkit 13.0.0 archive. URL: https://docs.nvidia.com/cuda/archive/13.0.0/hopper-tuning-guide/index.html (visited on 10/21/2026).

[22] PyTorch contributors. *Custom C++ and CUDA Operators*. Online tutorial. URL: https://docs.pytorch.org/tutorials/advanced/cpp_custom_ops.html (visited on 10/21/2026).

[23] NVIDIA. *CUDA Compatibility*. Online documentation. URL: https://docs.nvidia.com/deploy/cuda-compatibility/ (visited on 10/21/2026).

[24] NVIDIA. *Compute Sanitizer User Manual*. Online documentation. URL: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html (visited on 10/21/2026).

[25] NVIDIA. *NCCL User Guide*. Online documentation. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html (visited on 10/21/2026).

[26] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009). Metadata checked via Crossref; publisher full text not retrieved, pp. 65–76. DOI: 10.1145/1498765.1498785.

[27] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. arXiv v2; preprint version checked. 2024. arXiv: 2407.08608. URL: https://arxiv.org/abs/2407.08608v2 (visited on 10/21/2026).

[28] NVIDIA. *CUDA Stream Semantics: NCCL User Guide*. Online documentation. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/streams.html (visited on 10/21/2026).